NECパーソナルコンピュータ
PC-9800シリーズ

Software Library

MS-DOS® 3.3D

プログラマーズリファレンスマニュアル Vol.1

Software
Library
MS-DOS® 3.3D
プログラマーズリファレンス
マニュアル Vol.1

ご注意

(1) 本書の内容の一部または全部を、無断で他に転載することは禁止されています。
(2) 本書の内容は、将来予告なしに変更することがあります。
(3) 本書の内容は、万全を期して作成しております。万一、ご不審な点や誤り、記載もれなどお気づきの点がありましたら、ご連絡ください。
(4) 運用した結果の影響については、(3)項にかかわらず責任を負いかねますのでご了承ください。

Microsoft（マイクロソフト）とそのロゴは米国マイクロソフト社の登録商標です。
MS-DOS は米国マイクロソフト社の登録商標です。
Intel（インテル）は米国インテル社の商標です。
8086、80286は米国インテル社の商標です。

輸出する際の注意事項

本製品（ソフトウェア）は日本国内仕様であり、外国の規格等には準拠しておりません。
本製品を日本国外で使用された場合、当社は一切責任を負いかねます。また、当社は本製品に関して、海外での保守サービスおよび技術サポート等は行っておりません。

AG1A1A

はじめに

MS–DOSプログラマーズリファレンスマニュアル（Vol.1/Vol.2）は、システムプログラマーの方のために、MS–DOSのもとで動作するプログラムを開発する際に必要な、MS–DOSの技術情報を提供するものです。

本書では、プロセス管理やメモリ管理などのMS–DOS本体の技術資料と、MS–DOSでプログラマーが利用することができる各種のシステムコールやファンクションリクエストについて説明しています。

なお、このマニュアルを十分に利用していただくためには、ソフトウェアおよびハードウェアに関してある程度の専門的な知識が必要となります。

本書の目的と構成

本書は1章から5章で構成されています。

■ 第1章「システムコール」

MS–DOSで使用できる割り込みとシステムコールを、用例とともに説明しています。

■ 第2章「拡張機能」

PC–9800本体に用意された、拡張機能の利用方法について説明しています。

■ 第3章「MS–DOS技術資料」

ディスクアロケーションについての技術資料です。

■ 第4章「MS–DOSコントロールブロックとワークエリア」

コントロールブロックとワークエリアについての技術資料です。

■ 第5章「プログラムヒント」

プログラム作成に役立つヒントを説明しています。

■ 付録A「EXEファイルの構造とローディング」

リンカユーティリティによって生成されたEXE形式ファイルの構造について説明しています。

■ 付録 B「インテルオブジェクトモジュールフォーマット」

8086 マイクロプロセッサのオブジェクト言語のフォーマットについて説明しています。

■ 付録 C「各種コード表」

プログラム作成時に役立つ各種コード表を掲載しています。

その他のマニュアル

「MS–DOS 拡張機能セット」には、本書の他に次のようなマニュアルが添付されています。

■ 『MS–DOS ユーザーズリファレンスマニュアル』

システムディスクに収められている MS–DOS のすべてのコマンドについて、詳しく説明しています。また、「MS–DOS 基本機能セット」では扱われていない、MS–DOS の高度な機能についても解説しています。MS–DOS の手引きとして、ご利用ください。

■ 『日本語入力ガイド』

MS–DOS 上で利用可能な日本語入力機能について解説しています。日本語の入力を行う方法と、その他の有用な機能について詳しく説明し、また、辞書ファイルを保守管理するユーティリティ（DICM）や、ユーザーが独自の記号や漢字を作成して利用するためのユーティリティ（USKCGM）についても説明しています。

■ 『プログラム開発ツールマニュアル』

「MS–DOS プログラム開発ツールディスク」に収められているユーティリティプログラムの、詳細な使用法について解説しています。アセンブリ言語などでプログラムを開発される際に、ご利用ください。

■ 『プログラマーズリファレンスマニュアル Vol.2』

オペレーティングシステムの構成要素である MS–DOS デバイスドライバについての説明と、いくつかの周辺装置を制御するデバイスドライバの技術情報を提供しています。

目　次

第 1 章

システムコール

1.1 イントロダクション

MS–DOSでは、システムの操作・管理や入出力、各種のサービスをサブルーチンとして提供しています。これらのサブルーチンはシステムコールといい、ユーザーはアプリケーションプログラムから決められた手続きに従って、これらを利用することができます。

システムコールを利用してプログラムを作成すれば、さまざまな機能を簡単に利用することができ、また、MS–DOSのシステムコールはどの機種もほとんど共通なので、容易に他機種への移植が可能となります。そのうえ、MS–DOSの将来のバージョンでも問題なく動作する可能性が高くなります。

MS–DOSのシステムコールは、ソフトウェア割り込みを使って利用します。通常、MS–DOSで使用する割り込みタイプは、20H～27Hと、予約されている28H～3FHです。

割り込みタイプ21Hは、とくに〝ファンクションリクエスト〟と呼ばれ、MS–DOSがサポートするほとんどの機能を利用することができます。

本書では、MS–DOSシステムコールを次のように分類して解説します。

標準キャラクタデバイスI/O
メモリ管理
プロセス管理
ファイルとディレクトリの管理
MS–Networks
その他のシステムコール

1.2 標準キャラクタデバイスI/O

標準キャラクタのファンクションリクエストを使うと、コンソール、プリンタ、シリアルポートなどのキャラクタデバイスに対して、すべて同じ手続きで入出力させることができます。

次の表は、標準キャラクタデバイス入出力のファンクションリクエストの一覧です。これらのファンクションリクエストを用いると、入出力のリダイレクトもできます。

コード	機　能
01H	標準入力から1文字受け取り、その文字を標準出力に出力する
02H	標準出力に1文字出力する
03H	補助入力装置から1文字受け取る
04H	補助出力装置に1文字出力する
05H	プリンタに1文字出力する
06H	標準入力から1文字受け取る。または、標準出力に1文字出力する
07H	標準入力から1文字受け取る
08H	標準入力から1文字受け取る。受け取った文字の出力はしない
09H	標準出力に文字列を出力する
0AH	標準入力から文字列を受け取る
0BH	標準入力のバッファの状態を返す
0CH	標準入力のバッファを空にして、標準入力から1文字受け取る

表中の一部のファンクションリクエストは同じ機能をもっていますが、キャラクタを標準入力から標準出力にエコーするか、コントロールキャラクタをチェックするかどうかなど、細かい違いがあります。この違いの詳細は、ファンクションリクエストの個々のリファレンスを参照してください。

1.3　メモリ管理

MS–DOSは、各プロセスが使用しているメモリ領域の先頭に設定されたメモリコントロールブロックによって、メモリの割り当てを管理しています。このメモリコントロールブロックには、そのメモリ領域が使われているかどうか、使用中ならばそのメモリブロックを要求したプロセスのPSP（プログラムセグメントプレフィクス）のセグメントアドレス、メモリコントロールブロックが管理するメモリブロックのサイズなどが書き込まれています。あるメモリ領域が使われていなければ、他のプロセスで使うことができます。

次の表はメモリ管理のMS–DOSファンクションリクエストの一覧です。

コード	機　能
48H	メモリブロックの割り当てを要求する
49H	割り当てられたメモリブロックを開放する
4AH	割り当てられたメモリブロックを変更する

プロセスがファンクション48Hによってメモリの割り当てを要求すると、MS–DOSは要求を満たす大きさの空きメモリブロックを捜します。条件に見合う空きメモリブロックが見つかると、MS–DOSはそのメモリコントロールブロックを書き直し、要求を出したプロセスの所有するメモリとします。

空きメモリブロックが要求量よりも大きいと、メモリコントロールブロックのサイズフィールドを要求量に合うように修正し、必要量をプロセスに割り当てます。次に、残った空きメモリ領域の先頭に、新しいメモリコントロールブロックを作成し、ポインタを更新して、このメモリブロックをメモリコントロールブロックのチェイン（連鎖）に加えます。そして、MS–DOSはメモリを要求したプロセスへ、

割り当てたメモリブロックの先頭バイトのセグメントアドレスを返します。

プロセスがファンクション 49H によってメモリブロックを開放すると、MS–DOS はメモリコントロールブロックを、いずれのプロセスにも所有されていない利用可能なものとします。

プロセスがファンクション 4AH を使って、メモリブロックサイズを縮小させると、MS–DOS は、サイズ縮小によって開放されたメモリ領域の先頭にメモリコントロールブロックを作成し、メモリコントロールブロックのチェインに加えます。

プロセスがファンクション 4AH を使って、メモリブロックサイズを拡大させると、MS–DOS は、メモリブロックを割り当てるときと同様に扱いますが、セグメントアドレスは返さず、追加メモリブロックとそれまでのメモリブロックをチェインします。

ファンクション 48H または 4AH で、要求量を満たす空きメモリブロックが見つからないと、MS–DOS はメモリを要求したプロセスにエラーコードを返します。

プログラム（プロセス）は制御が渡されたら、まずファンクション 4AH によって、PSP から始まるメモリアロケーションブロックの初期設定値を修正し、ブロックを必要なだけの大きさに切り詰めるとよいでしょう。この処置によって不要なメモリ領域を開放し、資源を節約できます。また、この処置のあるプログラムは、将来マルチタスク処理がサポートされた場合、移植性の高いものになります。

プログラムが EXIT（終了）すると、呼び出したプログラム（アプリケーションを呼び出すのは通常 COMMAND.COM）がコントロールを取り戻す前に、MS–DOS が自動的にメモリアロケーションブロックを開放します。プロセスが EXIT することによって、MS–DOS はそのプロセスが占有していたメモリをすべて開放します。

どのようなプログラムも、メモリコントロールブロックをこわしてはなりません。もしメモリコントロールブロックのチェインが破壊されると、メモリアロケーションエラーとなり、システムを再起動しなければなりません。

1.4　プロセス管理

MS–DOS はプログラムのロード、実行、終了などのプロセスに関する種々のシステムコールを備えています。アプリケーションプログラムからでも、これらのシステムコールを使って他のプログラムの管理ができます。

次の表は、プロセス管理のための MS–DOS ファンクションリクエストの一覧です。

コード	機　能
31H	プログラムをメモリ中に常駐させたまま終了させ、呼び出したプログラムに制御を返す
4B00H	プログラムをロードし、実行する
4B03H	プログラム（オーバーレイ）をロードするが、実行しない
4CH	呼び出したプログラムに制御を返す
4DH	子プロセスが EXIT したときのリターンコードを返す
62H	カレントプロセスのプログラムセグメントの先頭セグメントアドレスを返す

■ プログラムのロードと実行

ファンクション 4B00H によって、あるプログラムが別のプログラムを起動すると、次の手順で処理されます。

まず、MS–DOS によってメモリが割り当てられます。次に、割り当てられたメモリの先頭（オフセット 0000H）に、新しいプログラムセグメントプレフィクス（PSP）が書き込まれます。続いて、プログラムがロードされ、プロセスの制御が目的のプログラムに渡されます。ファンクション 4CH によって、呼び出されたプログラムが EXIT（終了）すると、呼び出したプログラムに制御が返されます。

COMMAND.COM は、ファンクション 4B00H を使ってコマンドをロードし、実行しています。アプリケーションでも同様にプロセス管理をすることができ、子プロセスをメモリの許す限り実行することができます。

MS–DOS で実行可能なプログラムには、COM 形式（.COM の拡張子をもつ）と EXE 形式（.EXE の拡張子をもつ）の 2 種類の形式があります。これまでの解説は、両形式に共通です。次に、両者の違いは次のとおりです。

● COM 形式のロードと実行

COMMAND.COM は、COM 形式のプログラムをロードし実行するとき、すべての空きメモリ領域をアプリケーションに割り当て、64K バイト以上のメモリをプログラムに割り当てることができれば、オフセット 0000H を SP にセットし、スタックに 0 を PUSH して SP = FFFEH とします。割り当てるメモリが 64K バイトよりも少ないときは、プログラムの最上位オフセット＋1 を SP にセットし、0 を PUSH します。

COM 形式のプログラムは、ファンクション 4AH によって、メモリアロケーションブロックの初期値が縮小される前にスタック領域を確保します。なぜなら、既定のスタック領域は、開放されるメモリ領域にあるからです。

もし、新たにロードされたプログラムが、COM 形式のプログラムのように、すべてのメモリを割り当てられたり、ファンクション 48H によって空き領域のすべてを要求すると、MS–DOS は COMMAND.COM の非常駐部分も割り当てます。

プログラムがこの領域を変化させて終了すると、MS–DOS は COMMAND.COM の非常駐部を再ロードしてから、制御を COMMAND.COM に戻します。

もし、プログラムがメモリを十分に開放せずに常駐終了すると（ファンクション 31H）、COMMAND.COM の非常駐部を再ロードできず、システムが停止する危険があります。COM 形式のプログラムでは、このような事態の発生を最小限に抑えるため、事前にファンクション 4AH を使って、分割されるブロックの初期値を小さくしてください。また、プログラムが常駐終了する前に、ファンクション 49H によって、不必要なメモリはすべて開放するようにしてください。

● EXE 形式のロード方法

COMMAND.COM は、EXE 形式のプログラムのロードと実行を、次の手順で行います。

まず、EXE 形式のプログラム自体のサイズ（メモリイメージ）によって、そのプログラムが必要とするメモリ量を確保します。このメモリ量は、メモリが十分に

あるとき、ファイルヘッダの MAXALLOC のフィールド（オフセット 0CH）の値、足りないときは MINALLOC フィールド（オフセット 0AH）の値です。これらのフィールドの値は、リンカによって設定されています。

次に、MS–DOS はファイルヘッダの情報によって、EXE 形式のファイルの実アドレスを決定し、プログラムをロードします。

その後、制御がプログラムに引き渡されます。

MS–DOS における、COM 形式と EXE 形式のプログラムのロードの詳細については、第 3 章「MS–DOS 技術資料」と第 4 章「MS–DOS コントロールブロックとワークエリア」を参照してください。

●プログラムから別のプログラムを実行する方法

COMMAND.COM はパスを設定したり、パスを使って実行可能なプログラムを捜したり、EXE 形式のファイルをリロケート（再配置）するなどの細かい処理まで行います。したがって、あるプログラムから別のプログラムを実行するには、COMMAND.COM をコピーして使い、COMMAND.COM の実行を通して別のプログラムのロードや実行をする方法が最も簡単です。

これは、コマンドラインに/C スイッチをつけ、目的のプログラムを起動する方法です（詳しくはファンクション 4B00H の解説を参照してください）。

■ オーバーレイのロード

ファンクション 4B03H を使って、オーバーレイ形式のプログラムをロードするとき、プログラムは、オーバーレイ部分がロードされるセグメントアドレスを MS–DOS に知らせなければなりません。プログラムはオーバーレイをコールするとき、ロードするセグメントアドレスを指定し、オーバーレイはコールしたプログラムへディレクトリを返します。オーバーレイをコールしたプログラムは、これらの手順を完全に管理する必要があります。MS–DOS はオーバーレイに対して PSP を書き込んだり、他の方法で干渉することはありません。

MS–DOS は、コールしたプログラムのもっている（使っている）メモリ領域にオーバーレイがロードされても、そのことをチェックしません。

もし、コールしたプログラムが十分なメモリをもたずにオーバーレイをロードすると、メモリコントロールブロックがこわれ、メモリアロケーションエラーが生じます。このときはシステムを再起動するしかありません。

このため、オーバーレイをロードするプログラムは、ファンクション 4AH を使ってメモリアロケーションブロックの初期値を縮小するとき、必ずオーバーレイを格納する場所を用意するか、メモリアロケーションブロックの初期値を最小に縮小してから、ファンクション 48H を使ってオーバーレイのためにメモリを割り当てるようにしてください。

1.5　ファイルとディレクトリの管理

MS–DOS のファイルを扱うには、ハンドルと FCB（ファイルコントロールブロック）を用いる方法の2種類あります。ここでは、ハンドルによるファイルの扱いと階層ディレクトリ構造を利用するためのファンクションリクエストについて解説します。FCB については、1.8「バージョン 2.0 以前のシステムコール」で解説します。

■ ハンドル

ファイルを作成したりオープンするためには、パス名やファイルを割り付けるための属性をパラメータとしてファンクションリクエストを使用します。これで、ハンドルと呼ばれる 16 ビットの数字が返されます。以後は、このハンドルを利用してファイルの読み書きなどを行います。

ハンドルは、ディスク上のファイルあるいはデバイスファイルのどちらかと対応します。MS–DOS では、デバイスファイルのために5つの標準ハンドルを設定しています。これらは常にオープンされているので、使用するときオープンする必要はありません。次の表はその一覧です。

ハンドル	標準デバイス名
0	標準入力　（リダイレクト可）
1	標準出力　（リダイレクト可）
2	エラー出力
3	補助装置
4	プリンタ

ファイルを作成したりオープンするときに、利用可能な最初のハンドルが割り付けられます。1つのプログラムがオープンできるハンドルの数は 20 で、この中には先の5つの標準デバイスが含まれるため、通常1つのプログラムでオープンできるファイルの数は 15 です。5つの標準デバイスのいずれも、ファンクション 46H（ファイルハンドルの強制二重化）を使って、ファイルやデバイスを一時的に連結させることができます。

■ ファイル管理のファンクションリクエスト

MS–DOS は、ファイルを単純なバイト列として扱います。したがって、レコード構造やそれに関する特別なアクセス方法はありません。ファイルの読み出し／書き込みには、データバッファへのポインタと読み書きするバイト数だけを必要とします。

次の表はファイル管理のファンクションリクエストです。

コード	機　能
3CH	ハンドルを使ってファイルを作成する
3DH	ハンドルを使ってファイルをオープンする
3EH	ファイルをクローズする
3FH	読み出しファイルかデバイスから読み出す
40H	書き込みファイルかデバイスへ書き込む
42H	読み書きするファイル中のポインタを移動する
45H	新規のハンドルをオープンし、すでにオープンされている他のハンドルと連結させる
46H	すでにオープンされているハンドルと、すでにオープンされている他のハンドルとを、強制的に連結させる
5AH	一時ファイルを他のファイルと重複しない名で作成する
5BH	新しいファイルを作成する。ただし、同じファイル名が存在するときは、ファイルを作成しない
67H	ひとつのプログラムがアクセスできる最大ハンドル数を設定する
68H	ファイルをクローズせずに、バッファ中のデータをクリアする
6CH	ハンドル付きファイルをオープン／新規作成して、さらにバッファ中のデータをクリアする

■ ファイルシェアリング

MS–DOS バージョン 3.1 以降では、1 つ以上のプロセスがファイルを共有してアクセスできる、ファイルシェアリングシステムを導入しています。ファイルシェアリングは、ファイルシェアリングをサポートするシェアリングコマンドが実行され、SHARE.EXE がメモリに常駐すると有効となります。次の表は、ファイルシェアリングで使われるファンクションリクエストの一覧です。

コード	機　能
3DH	ファイルシェアリングモードにして、1 つのファイルをオープンする
440BH	致命的なエラーが発生したとき、割り込みタイプ 24H を実行する前にリトライ（再試行）する回数を設定する
5C00H	ファイルの一部をロックする
5C01H	ファイルの一部のロックを解除する

ファイルシェアリングが有効でないと、これらのファンクションリクエストは使用できません。ファンクション 3DH（ハンドルを使うファイルのオープン）は、種々のモードで動作します。コンパチビリティモードでは、ファイルシェアリングが有効でなくても使えます。ファイルシェアリングモードでは、ファイルシェアリングが有効なときだけ使うことができます。

■ デバイス管理のファンクションリクエスト

ファンクション 44H は、デバイスへの I/O コントロールを実行します。このファンクションリクエストは、異なるデバイスを扱う種々のコードを含んでいま

す。一部の IOCTL ファンクションリクエストは、デバイスドライバが IOCTL ファンクションをサポートするために使用されます。次の表は、MS–DOS のデバイス管理のファンクションリクエストの一覧です。

コード	機　能	説　　明
4400H	IOCTL データを取得する	デバイスの種類を取得する
4401H	IOCTL データを設定する	デバイスの種類を設定する
4402H	IOCTL をキャラクタデバイスから受け取る	キャラクタデバイスからコントロールデータを受け取る
4403H	IOCTL をキャラクタデバイスへ送る	キャラクタデバイスへコントロールデータを送る
4404H	IOCTL をブロックデバイスから受け取る	ブロックデバイスからコントロールデータを受け取る
4405H	IOCTL をブロックデバイスへ送る	ブロックデバイスへコントロールデータを送る
4406H	入力ステータスのチェック	デバイスの状態が入力かどうかをチェックする
4407H	出力ステータスのチェック	デバイスの状態が出力かどうかをチェックする
4408H	媒体が交換可能か調べる	ブロックデバイスが差し換え可能な媒体かどうかをチェックする
440CH	一般 IOCTL（ハンドル用）	プリンタに対して出力繰り返し回数の設定と取得をする
440DH	一般 IOCTL（ブロックデバイス用）	ブロックデバイスへのデバイスパラメータの設定と取得を行う
440EH	論理ドライブマップの取得	現在の論理ドライブと物理デバイスのマップ情報を取得する
440FH	論理ドライブマップの設定	論理ドライブを物理ドライブにマップする

一部の IOCTL ファンクションリクエストの形式は、MS–Networks でしか使えません。詳しくは、1.6「MS–Networks」を参照してください。

■ ディレクトリ管理のファンクションリクエスト

ディスクのルートディレクトリが管理できるディレクトリとファイル名の数は、メディアの容量に制限されます。ハードディスクでのディレクトリとファイル名の数は、MS–DOS のパーティション容量に依存します。サブディレクトリは、特別な属性をもったファイルで、サブディレクトリ下に作成できるディレクトリとファイルの数は、ディスクの空き容量だけに制限されます。パス名の長さは、64

文字（半角の英数字）を超えることはできません。

サブディレクトリはバージョン 2.0 からサポートされたものです。バージョン 2.0 以前の MS–DOS で作成されたディスクは、単にルートディレクトリだけをもつものとして扱われます。

次の表はディレクトリ管理のファンクションリクエストの一覧です。

コード	機　能
39H	サブディレクトリを新規作成する
3AH	サブディレクトリを削除する
3BH	カレントディレクトリを変更する
41H	ディレクトリエントリ（ファイル）を削除する
43H	ファイルの属性の設定と取得をする
47H	指定ドライブのカレントディレクトリを返す
4EH	該当するファイル（ワイルドカード等で指定した）を検索する
4FH	該当するファイル（ワイルドカード等で指定した）の検索を続行する。このファンクションはファンクション 4EH に続いて実行される
56H	ディレクトリエントリ（ファイル）名を変更する
57H	ファイルの日付または時刻を、設定または取得する

■ ディレクトリエントリ

ディレクトリエントリは、ファイル名、最後に変更された日付と時刻、ファイルサイズなどを含む 32 バイトのレコードです。サブディレクトリのエントリはルートディレクトリのエントリと同じです。ディレクトリエントリについての詳細は、第 3 章「MS–DOS 技術資料」を参照してください。

■ ファイルの属性

次の表は、ファイルの属性（アトリビュート）とディレクトリエントリの属性を表すバイト（オフセット 0BH）の一覧です。属性は、ファンクション 43H によって調べたり、変えることができます。

コード	内　容
00H	通常のファイル。自由に読み出しや書き込みができる
01H	読み出し専用。書き込むためにファイルをオープンにすることも、同じ名前のファイルを作成することもできない
02H	隠しファイル。DIR コマンドでは見ることができない
04H	システムファイル。DIR コマンドでは見ることができない
08H	ボリューム ID。この属性をもてるファイルは、ルートディレクトリ上に 1 つだけ存在する
10H	サブディレクトリ
20H	アーカイブ（保存）ファイル。ファイルが変更されたときに作られる

ボリューム ID（08H）とディレクトリ（10H）の属性は、ファンクション 43H では変更できません。

1.6 MS–Networks

MS–Networks は、1つ以上のサーバとワークステーションから構成されます。MS–DOS は、サーバに対するワークステーションドライブとワークステーションデバイスの割り当ての情報を保管します。MS–Networks の詳細は、MS–Networks マネージャーズガイドとユーザーズガイドを参照してください。

次の表は、MS–Networks 管理のファンクションリクエストの一覧です。

コード	機　能	説　　明
4409H	リモートブロックデバイスの検出	ドライブ名によって Networks のワークステーションか、サーバかを調べる
440AH	リモートハンドルの検出	ハンドル名によって Networks のワークステーションか、サーバかを調べる
5E00H	マシン名を取得する	ワークステーションの Networks 名を取得する
5E02H	プリンタセットアップ	Networks プリンタへ送るファイルの先頭に、コントロールキャラクタをセットする
5F02H	割り当てリストのエントリを取得する	Networks の割り当てリストのエントリ（ワークステーションのドライブ名またはデバイス名、再割り当てされたディレクトリやデバイスの Networks 名など）を取得する
5F03H	割り当てリストのエントリを作成	ワークステーションのドライブやデバイスから、サーバへリディレクションを行う
5F04H	割り当てリストのエントリの取り消し	ワークステーションからサーバへのリディレクションを取り消す

1.7 その他のシステムコール

システムコールは、これまで述べてきたもの以外に、ドライブ、クロック、アドレスなどのシステムの情報を管理します。

次の表は、種々のシステムを管理するMS–DOSファンクションリクエストの一覧です。

コード	機　能	説　　明
0DH	ディスクのリセット	ファイルバッファを空にする
0EH	ディスクの選択	デフォルトドライブを設定する
19H	カレントドライブの取得	カレントドライブの番号を返す
1AH	ディスク転送アドレスの設定	ディスク転送バッファのアドレスを設定する
1BH	デフォルトドライブのデータの取得	デフォルトドライブのフォーマット情報を返す
1CH	ドライブのデータの取得	ディスクのフォーマット情報を返す
25H	割り込みベクタの設定	割り込み処理ルーチンのアドレスを設定する
29H	ファイル名の解析	ファイル名の文字列を解析する
2AH	日付の取得	システムの日付を取得する
2BH	日付の設定	システムの日付を設定する
2CH	時刻の取得	システムの時刻を取得する
2DH	時刻の設定	システムの時刻を設定する
2EH	ベリファイフラグのセット／リセット	ベリファイフラグをセット／リセットする
2FH	ディスク転送アドレスの取得	ディスク転送アドレスを取得する
30H	MS–DOSバージョン番号の取得	MS–DOSのバージョン番号を返す
33H	<CTRL–C>チェックのセット／リセット	<CTRL–C>チェックのステータスを返す
35H	割り込みベクタの取得	割り込みルーチンのアドレスを返す
36H	ディスクのフリースペースの取得	ディスクのフリースペースのデータを返す
38H	国別情報の設定と取得	国別情報の設定か取得をする
54H	ベリファイのステータスを返す	ベリファイのステータスを返す
65H	拡張国別情報の取得	拡張国別情報を返す
6601H	コードページの取得	デフォルト時と現在のコードページを返す
6502H	コードページの設定	コードページを設定する

1.8　バージョン 2.0 以前のシステムコール

MS-DOS は現在のバージョンでも、バージョン 2.0 以前の古いシステムコールをサポートしています。これは、バージョン 2.0 以前のプログラムとの互換性を保つためだけに残しているものです。

バージョン 2.0 以降には、バージョン 2.0 以前のシステムコールを代用するファンクションリクエストが用意されているので、プログラムを作成するときは、バージョン 2.0 以前のシステムコールを使わないようにしてください。次の表は、この対応の一覧表です。

Ver.2.0 以前のファンクションコール	Ver.2.0 以降、代用されるファンクションリクエスト
00H　プログラムの終了	4CH　プロセスの終了
0FH　ファイルのオープン	3DH　ハンドルを使うファイルのオープン
10H　ファイルのクローズ	3EH　ハンドルを使うファイルのクローズ
11H　最初のエントリを検索	4EH　最初に一致するファイル名の検索
12H　次のエントリを検索	4FH　次に一致するファイル名の検索
13H　ファイルの削除	41H　ディレクトリエントリの削除
14H　シーケンシャルな読み出し	3FH　ファイルかデバイスの読み出し
15H　シーケンシャルな書き込み	40H　ファイルかデバイスの書き込み
16H　ファイルの作成	3CH　ハンドルを使うファイルの作成
	5AH　一時ファイルの作成
	5BH　新しいファイルの作成
17H　ファイル名の変更	56H　ディレクトリエントリの変更
21H　ランダムな読み出し	3FH　ファイルかデバイスの読み出し
22H　ランダムな書き込み	40H　ファイルかデバイスの書き込み
23H　ファイルの大きさを取得する	42H　ファイルポインタの移動
24H　相対レコードの設定	42H　ファイルポインタの移動
26H　新しい PSP を作成する	4B00H　プログラムのロードと実行
27H　ランダムなブロックの読み出し	3FH　ファイルかデバイスの読み出し
28H　ランダムなブロックの書き込み	40H　ファイルかデバイスの書き込み
Ver.2.0 以前のシステムコール	**代用されるファンクションリクエスト**
20H　プログラムの終了	4CH　プロセスの終了
27H　プログラムの常駐終了	31H　プロセスの常駐終了

■ ファイルコントロールブロック

バージョン 2.0 以前のファイル管理のファンクションリクエストは、ファイルのファイルコントロールブロック（FCB）をアクセスします。この FCB は、ファイル名、サイズ、レコード長、カレントレコードのポインタなどの情報を含んでいます。新しいハンドル形式のファンクションリクエストのほとんどのファイル操作は、FCB 形式のファンクションリクエストでも実行できます。

PSP 内のオフセット 5CH と 6CH に、2 つの FCB のための空き領域が用意されています。FCB を取り扱うバージョン 2.0 以前のシステムコールでは、〝オープンされている FCB〟、〝オープンされていない FCB〟のアドレスを、指定するレジスタにセットします。オープンされていない FCB とは、ドライブ名とファイル名だけが入っているもので、ワイルドカード文字（*、?）を入れることができます。オープンされた FCB のすべてのフィールドは、オープンファイルシステムコール（ファンクション 0FH）によって埋められます。PSP の説明と FCB の利用法については、第 4 章「MS-DOS ファイルコントロールとワークエリア」を参照してください。次の表は、FCB のフィールドの内容を示します。

フィールド名	大きさ（バイト）	オフセット	
		16 進	10 進
ドライブ番号	1	00H	0
ファイル名	8	01H～08H	1～8
拡張子	3	09H～0BH	9～11
カレント（現在の）ブロック	2	0CH, 0DH	12, 13
レコードサイズ	2	0EH, 0FH	14, 15
ファイルの大きさ	4	10H～13H	16～19
最後の書き込みが行われた日付	2	14H, 15H	20, 21
最後の書き込みが行われた時刻	2	16H, 17H	22, 23
予約域	8	18H～1FH	24～31
カレント（現在の）レコード	1	20H	32
相対レコード	4	21H～24H	33～36

■ FCB のフィールド

オフセット　00H：ドライブ番号

ディスクドライブを指定します。1 はドライブ A、2 はドライブ B、…を指定します。FCB をファイルの作成またはオープンのために使用するとき、このフィールドを 0 に設定すると、カレントドライブ（現在アクセスできるドライブ）を指定することができます。オープンファイルシステムコール（ファンクション 0FH）を行うと、このフィールドを実際のドライブの番号に設定することができます。

オフセット　01H：ファイル名

8 文字までの長さのファイル名を設定できます。ファイル名はフィールドの先頭から入り、8 文字に満たない場合はスペースが入ります。予約されたデバイス

ファイル（PRNなど）を指定するとき、コロン（:）をファイル名の最後に付けないでください。

オフセット　09H：ファイル名拡張子

フィールドの先頭から、3文字までの長さのファイル名拡張子が入り、3文字に満たないときはスペースが入ります。拡張子がないときは、すべてスペースになります。

オフセット　0CH：カレントブロック

現在のレコードが入っているブロック（128レコードが1単位）を示すポインタです。このカレントブロックフィールドとカレントレコードフィールド（オフセット20H）によって、目的のレコードポインタを作成します。このフィールドは、オープンファイルシステムコールによって0に設定されます。

オフセット　0EH：レコードサイズ

バイト単位で表した論理レコードの長さを設定します。オープンファイルシステムコールによって、128が設定されます。レコード長が128バイトでないと、ファイルをオープンした後、このフィールドを設定しなければなりません。

オフセット　10H：ファイルのサイズ

バイト単位で表すファイルの大きさです。このフィールドの先頭の2バイトはファイルの大きさの下位2バイトに、残りの2バイトがファイルの大きさの上位2バイトになります。

オフセット　14H：最後に書き込みが行われた日付

ファイルが作成、更新された日付は次のように2バイトに設定されます。

年は0〜99（1980〜2079）が設定されます。

オフセット 15H								オフセット 14H							
15						9	8			5	4				0
Y	Y	Y	Y	Y	Y	Y	M	M	M	M	D	D	D	D	D
年							月				日				

オフセット　16H：最後の書き込みが行われた時刻

ファイルが作成、更新された時刻、分、秒は、次のように2バイトに設定されます。

オフセット 17H								オフセット 16H							
15				11	10					5	4				0
H	H	H	H	H	M	M	M	M	M	M	S	S	S	S	S
時					分						秒/2				

オフセット　18H：予約域

このフィールドは、MS–DOS が使用するために確保されています。

オフセット　20H：カレントレコード

現在のブロック内の 128 個のレコードのうちの 1 つを示します。前述のカレントブロックフィールド（オフセット 0CH）と、このカレントレコードフィールドによって、カレントレコードポインタが作成されます。オープンファイルシステムコールは、このフィールドの初期値設定をしません。このファイルに対してシーケンシャルなリード／ライトを行うには、事前にこのフィールドを設定しておかなければなりません。

オフセット　21H：相対レコード

ファイルの先頭（0 から始まる）からカウントした、現在選択されているレコード番号を示します。オープンファイルシステムコールは、このフィールドの初期値設定をしません。このファイルに対して、ランダムなリード／ライトを行うには、事前にこのフィールドを設定しておかなければなりません。レコードサイズが 64 バイト未満のときはフィールド全体の 4 バイトが、64 バイト以上のときは最初の 3 バイトのみが使用されます。

注意　PSP 内のオフセット 5CH の FCB を使用するとき、相対レコードフィールドの最終バイトは、オフセット 80H から開始するフォーマットされていないパラメータエリアの先頭バイトです。これは、デフォルトのディスク転送アドレスです。

■ 拡張 FCB

拡張 FCB は、ディスクディレクトリ中で、特別な属性をもつファイルを作成・検索するために使用されます。拡張 FCB は、通常、FCB の前の 7 バイトからなり、次のフォーマットになっています。属性バイトの詳細は、1.5「ファイルとディレクトリの管理」を参照してください。

フィールド名	大きさ（バイト）	オフセット（10 進）
フラグバイト（FFH） (拡張 FCB であることを示す。)	1	–7
予約域	5	–6
属性バイト	1	–1

1.9　システムコールの使い方

この章では、アプリケーションからシステムコールを使う方法を説明します。

■ 割り込みの使い方

MS–DOS は、システム自身が使用するために、20H から 3FH までの割り込みタイプを予約しており、80H～FCH に割り込みルーチンアドレステーブルをもっています。割り込みタイプの多くは、ファンクションリクエストに置き換えられています。なお、1.10「割り込み」では、ユーザーが 3 つの MS–DOS 割り込みハンドラ（プログラムの終了、<CTRL–C >、致命的エラーによる中断）ルーチンを作成するための解説をしています。

■ ファンクションリクエストの使い方

ファンクションリクエストとは、システム資源の管理を行う MS–DOS のルーチン群をコールするものです。ファンクションリクエストをコールする標準シーケンス（手続き）は次のとおりです。

1. 必要とするデータを、それぞれのレジスタに設定します。
2. ファンクション番号を、AH に設定します。
3. 必要ならば、アクションコードを AL に設定します。
4. 割り込みタイプ 21H を実行します。

もし、プログラムが標準のプログラムセグメントプレフィクス（PSP）をもっていると、割り込みタイプ 21H は、PSP 内のオフセット 50H をロングコールして代用できます。ただし、割り込みタイプ 21H の使用をおすすめします。

■ 高級言語からのコール

システムコールは、アセンブリ言語モジュールとリンク可能な高級言語から行うことができます。高級言語からのシステムコールは、次のように行います。

● C 言語からのコール

C 言語では多くの処理系が、システムコールを呼び出す機能をライブラリ関数として提供しています。詳しくはそれぞれの処理系の関数リファレンスを参考にしてください。

● BASIC からのコール

システムコールを利用するとき、コンパイラとインタプリタでは、異なる方法を使います。コンパイルされたモジュールは、アセンブリ言語で開発されたモジュールとリンクして 1 つのプログラムとすることができます。インタプリタの場合は、CALL 文または USR 関数を使用します。

■ レジスタの処理

MS-DOS は、ファンクションリクエストをコールしたとき、内部的にスタックを使います。このため、リターン情報として使われないレジスタの内容は保存されます。しかし、プログラムのスタック領域の大きさは、割り込み処理の実行に十分な大きさ（少なくとも、他の処理に必要な大きさ＋128 バイト）でなければなりません。

■ エラー処理

ファンクションリクエストは、エラーが起こるとキャリーフラグをセットし、AX にエラーコードを返します。次の表は、エラーコードの一覧です。

コード	意　味	
01H	ファンクションコードが無効	Invalid function code
02H	ファイルが見つからない	File not found
03H	パス名が見つからない	Path not found
04H	ファイルをオープンしすぎている	Too many open files(no open handles left)
05H	アクセスできない	Access denied
06H	ハンドルが無効	Invalid handle
07H	メモリコントロールブロックが破損	Memory control blocks destroyed
08H	メモリが足りない	Insufficient memory
09H	メモリブロックアドレスが無効	Invalid memory block address
0AH	環境が無効	Invalid environment
0BH	書式が無効	Invalid format
0CH	アクセスコードが無効	Invalid access code
0DH	データが無効	Invalid data
0EH	予約（使用されていない）	RESERVED
0FH	ドライブ名が無効	Invalid drive
10H	カレントディレクトリを削除しようとした	Attempt to remove the current directory
11H	同じデバイスではない	Not same device
12H	これ以上ファイルはない	No more files
13H	ディスクがライトプロテクトされている	Disk is write-protected
14H	ディスクユニットが不良	Bad disk unit
15H	ドライブが準備されていない	Drive not ready
16H	ディスクコマンドが無効	Invalid disk command
17H	CRC エラー	CRC error
18H	長さが無効	Invalid length(disk operation)
19H	シークエラー	Seek error
1AH	MS-DOS のディスクではない	Not an MS-DOS disk

コード	意味	
1BH	セクタが見つからない	Sector not found
1CH	紙切れ	Out of paper
1DH	書き込みが失敗	Write fault
1EH	読み出しが失敗	Read fault
1FH	一般的な失敗	General failure
20H	共有違反	Sharing violation
21H	ロック違反	Lock violation
22H	ディスクが不正	Wrong disk
23H	FCB使用不可能	FCB unavailable
24–31H	予約（使用されていない）	RESERVED
32H	ネットワークリクエストがサポートされていない	Network request not supported
33H	リモートコンピュータがLISTEN状態でない	Remote computer not listening
34H	ネットワーク名が重複している	Duplicate name on network
35H	ネットワーク名が見つからない	Network name not found
36H	ネットワークビジー	Network busy
37H	ネットワークデバイスはこれ以上ない	Network device no longer exists
38H	ネットワーク BIOSの限界を越えた	Net BIOS command limit exceeded
39H	ネットワークアダプタのハードエラー	Network adapter hardware error
3AH	ネットワークから不正な応答があった	Incorrect response from network
3BH	予期できないネットワークエラー	Unexpected network error
3CH	リモートアダプタが不正	Incompatible remote adapter
3DH	プリント待ち行列が一杯	Print queue full
3EH	待ち行列が一杯ではない	Queue not full
3FH	プリントファイルのためのスペースが足りない	Not enough space for print file
40H	ネットワーク名はすでに削除されている	Network name was deleted
41H	アクセスできない	Access denied
42H	ネットワークデバイスのタイプが不正	Network device type incorrect
43H	ネットワーク名が見つからない	Network name not found
44H	ネットワーク名の限界を越えた	Network name limit exceeded
45H	ネットワーク BIOSセッションの限界を超えた	Net BIOS session limit exceeded

コード	意　味	
46H	一時休止	Temporarily paused
47H	ネットワークの要求が受けつけられない	Network request not accepted
48H	プリンタかディスクのリダイレクト休止	Print or disk redirection is paused
49–4FH	予約（使用されていない）	RESERVED
50H	ファイルが存在する	File exists
51H	予約	RESERVED
52H	作成不能	Can not make
53H	割り込みタイプ 24H の失敗	Interrupt 24H failure
54H	ネットワーク構造が不正	Out of structures
55H	割り当て済み	Already assigned
56H	パスワードが無効	Invalid password
57H	パラメータが無効	Invalid parameter
58H	ネットワークへの書き込み失敗	Net write fault
5AH	システム関連ファイルがロードされていない	System component not loaded

エラーが発生すると、キャリーフラグがセットされ、エラーコードが存在する場合は、AX にエラーコードが返されます。エラー処理は、各コールのすぐ後に、次のステートメントを置きます。

JC　〈エラー処理ルーチンラベル〉

アプリケーションは AX の内容を調べ、エラーの内容を判別して処理ルーチンへ制御を移します。

拡張エラーコード

MS–DOS では、MS–DOS の古いバージョンとの互換性を維持するため、および古いエラーコードとの区別のために、初期の MS–DOS で使われていたエラーコードへ新規に追加されたものを、拡張エラーコードと呼んでいます。

これらの拡張エラーコードは、ファンクション 59H（拡張エラーコードを取得する）で得られ、MS–DOS が返す重大なエラーコードのほとんどを網羅しています。また、ファンクション 59H の項には、詳細なエラーコードの一覧と、このファンクションリクエストの使い方の解説があります。

■ システムコールの解説について

各システムコールの解説は、必要に応じて、実行前に設定するレジスタとその概要（コール）、実行後に返されるレジスタとその概要（リターン）、コールとリターンで使用するレジスタの詳細と機能の解説（解説）、マクロ定義の例（マクロ定義）、そのマクロを使ったプログラミング例（サンプル）の解説があります。

以下の図は、システムコールを行うときの各種レジスタのステータスの例として、ファンクション27H（ランダムなブロックの読み出し）の一部です。

INT 21H

ファンクション 27H　ランダムなブロックの読み出し

コール
AH　　＝27H
DS：DX＝オープンされたFCB
CX　　＝読み出すべきレコード数

リターン
AL ＝00H　読み出しは正常に行われ、処理が完了した
　 ＝01H　ファイルの終わり（EOF）。または空レコード
　 ＝02H　ディスク転送アドレス（DTA）に十分な空き領域がないため、読み出しは中止された。
　 ＝03H　ファイルの終わり（EOF）。レコードの残りの部分は、0で埋められた
CX ＝読み取られたレコード数

CXで指定したレコード数のデータを、ファイルからDTAに読み込みます。

■ サンプルプログラム

次のサンプルプログラムは、システムコールの使い方とデータの宣言だけから構成されています。このサンプルプログラムには、セグメントの宣言やMS-DOSへの戻り方といったプログラムを作る上での基本的なことが含まれています。なお、このサンプルプログラムは、COM形式のファイルとして実行されるものです。

```
code        segment
            assume  cs:code, ds:code, es:nothing, ss:nothing
            org     100H
start:      jmp     begin
;
filename    db      "b:¥extfile.asc", 0
buffer      db      129dup(?)
handle      dw      ?
;
begin:      open_handle     filename, 0         ;ファイルのオープン
            jc              error_open          ;エラー処理へ
            mov             handle, ax          ;ハンドルをセーブ
read_line:  read_handle     handle, buffer, 128 ;128バイトを読み込む
            jc              error_read          ;エラー処理へ
```

```
            cmp             ax, 0               ; ファイルエンドか?
            je              return              ; はいのとき、処理終了
            mov             bx, cx              ; いいえのとき、読み出すべき
                                                ; レコード数を設定
            mov             buffer[bx], "&"     ; 終了文字列の設定
            display         buffer              ; 文字列の表示 (09H)
            jmp             read_line           ; 読み込み処理を継続
return:     end_process     0                   ; 処理終了し MS-DOS へ戻る
last_inst:                                      ; メッセージ表示
;                                               ; プログラムの終了
code        ends
            end             start
```

このサンプルプログラムでは、システムコールの使い方をマクロ定義にしてあります。マクロ定義（open_handle、read_handle、display、end_process）については、ファンクションリクエストの各解説や第1章「システムコール」の章末にあるマクロ定義例を参照してください。

これらマクロは、第4章「MS–DOS コントロールブロックとワークエリア」で解説されている COM 形式のプログラムのための環境を想定しています。特別な例として、同じ値のすべてのレジスタを定義する場合があげられます。通常、マクロはレジスタの保護も、メインコードからエラー処理ルーチンに行くときのラベルのチェックも行いません。それらはマクロ定義のサブルーチンを小さくするために、マクロをコールするメインのアセンブルプログラムで定義します。

サンプルプログラムでのエラー処理

システムコールがエラーコードを返したとき、このサンプルプログラムはエラー状態をチェックし、エラー処理ルーチンへ移ります（エラールーチンの内容は省略します）。通常、エラー処理ルーチンは、簡単なメッセージを表示するだけで作業を続行します。しかし重大なエラーが起こると、メッセージを表示しプログラムを終了します（ただし、ファイルをクローズするなどの処理を行います）。

以下に、各割り込みタイプとシステムコールを解説します。

1.10　割り込み

プログラムで使用できる割り込みタイプ 20H〜27H について解説します。

16進	10進	機　　能
20H	32	プログラムの終了
21H	33	ファンクションリクエスト
22H	34	終了アドレス
23H	35	<CTRL-C>の抜け出しアドレス
24H	36	致命的エラーによる中断アドレス
25H	37	アブソリュートディスクリード
26H	38	アブソリュートディスクライト
27H	39	プロセスの常駐終了
28〜3FH	40〜63	予約

注意　各ファンクションリクエストの解説にあるサンプルプログラムは、参考のために記載しているものです。これらのサンプルプログラムは、そのままでは動作しません。

割り込みタイプ
20H プログラムの終了

コール CS ＝ PSP（プログラムセグメントプレフィクス）のセグメントアドレス

リターン なし

解説

現在のプロセスを終了し、制御を親プロセスに戻します。すべてのオープンされているファイルをクローズし、メモリを解放します。バージョン 2.0 以前の MS–DOS での COM ファイルの終了は、ほとんどこの割り込みで行われます。

この割り込みをかけるためには、その前に CS レジスタ内へ PSP のセグメントアドレスを入れておきます。

PSP 内のオフセットに設定されていた抜け出しアドレスは、以下のように MS–DOS に戻されます。詳しくはこの後の割り込みタイプ 22H から 24H の解説を参考にしてください。

抜け出しアドレス	オフセット	割り込みベクタ
プログラムの終了	0AH	INT 22H
<CTRL–C>	0EH	INT 23H
重大なエラー	12H	INT 24H

ファイルバッファの内容は、すべてディスクに書き出されます。

注意　この割り込みをかける前に、サイズを変更したすべてのファイルをクローズしてください。変更されたファイルがクローズされていないと、そのファイルの大きさはディレクトリに正しく書き込まれません。ファイルのクローズについては、ファンクション 10H、3EH を参照してください。

割り込みタイプ 20H は、MS-DOS バージョン 2.0 以前と互換性を保つために用意されているものです。バージョン 3.1 以降で開発する新規のプログラムは、ファンクションリクエスト 4CH を使用して、プロセスを終了させるようにしてください。

マクロ定義

```
terminate    macro
             int 20H
             endm
```

サンプル

この例は、画面にメッセージを表示し、MS-DOS に戻るプログラムです。1.9「システムコールの使い方」のサンプルプログラムも参照してください。

```
message      db "displayed by INT20H example", 0DH, 0AH, "$"
;
int_20H:     display     message ; 文字列の表示 (09H)
             terminate           ; プログラムの終了
code         ends
             end         start
```

割り込みタイプ

21H ファンクションリクエスト

コール

AH　=ファンクションリクエストの番号
AL　=サブファンクションの番号
他のレジスタ　=個々のファンクションで要求されるパラメータ

リターン

各ファンクションの解説を参照

解　説

各システムコールの呼び出しと実行を行います。AHレジスタには、目的のシステムファンクションの番号を、他のレジスタではシステムコールに渡すパラメータの設定をします。詳細については、1.11「ファンクションリクエスト」のリファレンスで解説します。

マクロ定義

個々のファンクションリクエストのマクロ定義については、1.11「ファンクションリクエスト」のリファレンスを参照してください。

サンプル

```
mov     ah, 2CH          ; 時刻を得るファンクション 2CH をコール
int     21H              ; ファンクションリクエスト
```

割り込みタイプ

22H 終了アドレス
23H 〈CTRL-C〉の抜け出しアドレス
24H 致命的エラーによる中断アドレス

解　説

これらは真の割り込みではなく、セグメントとオフセットアドレスのための記憶域の位置で、指定された環境下でMS-DOSによって割り込みがかけられます。ユーザーが独自に割り込みハンドラを作成した場合、ファンクションリクエスト35H（割り込みベクタを得る）を使ってアドレスを取得し、次にファンクションリクエスト25H（割り込みベクトの設定）を使って設定します。

割り込みタイプ22H……終了アドレス

プログラムを終了するとき、MS-DOSはベクタテーブルの割り込みタイプ22Hのエントリアドレスに制御を移行します。このアドレスは、MS-DOSがプログラムセグメントを作成するとき、PSP内のオフセット0AHにコピーされます。

割り込みタイプ23H……<CTRL-C>の抜け出しアドレス

<CTRL-C>を入力すると、MS-DOSはベクタテーブルの割り込みタイプ23Hのエントリアドレスに制御を移します。このエントリアドレスは、MS-DOSがPSPを作成するとき、PSP内のオフセット0EHにコピーされます。

ユーザーが独自に<CTRL-C>ルーチンを作成するとき、以下の点に注意してください。

<CTRL-C>ルーチンですべてのレジスタの内容を保存すると、IRET命令で、このルーチンを終了し、プログラムを継続することができます。割り込み発生時、すべてのレジスタの内容は、MS-DOSがコールされたときの値に設定されます。IRETで戻るときにレジスタの値を保存するかぎり、MS-DOSのシステムコールの使用を含む<CTRL-C>ルーチンに制限はありません。

<CTRL-C>ルーチンは、ロングリターン（Far RET）を使うことによって、キャリーフラグから割り込み発生前のプログラムを、強制終了するか続行するかを決定することができます。MS-DOSはキャリーフラグがセットされていると、プログラムを強制終了させ、設定されていなければ、IRETによって戻ったときと同様にプログラムを続行します。

プログラムがファンクションリクエスト09H、0AH、バッファードI/Oのいずれかを実行中に、<CTRL-C>によってユーザーが作成した<CTRL-C>ルーチンに割り込むと、IRETでプログラムを続行させる

ことによって、入出力は次の行の先頭から再開されます。

プログラムがファンクション 4B00H（プログラムのロードの実行）を使うなどして、第 2 の PSP を作り、ベクタテーブルの<CTRL–C>のアドレスを変更する第 2 のプログラムを実行すると、MS–DOS は、第 1 のプログラムに制御が戻る前に、<CTRL–C>のアドレスを第 2 のプログラムの実行前の値に戻します。

注意　MS–DOS は、INT23H を実行するとき、必ず画面に゛^C″と 0DH、0AH（キャリッジリターン、ラインフィード）を出力しますが、これを取り消すことはできません。

割り込みタイプ 24H……致命的エラーによる中断アドレス

I/O ファンクションコールの 1 つを実行しているとき、致命的ディスクエラーが発生すると、MS–DOS はベクタテーブルの割り込みタイプ 24H のエントリアドレスに制御を移します。このアドレスは、MS–DOS がプログラムセグメントを作成するとき、PSP 内のオフセット 12H にコピーされます。

割り込みタイプ 25H……アブソリュートディスクリード
割り込みタイプ 26H……アブソリュートディスクライト

これらの割り込みを実行中にエラーが発生した場合、割り込みタイプ 24H は実行できません。これらのエラーは、通常 COMMAND.COM 内の MS–DOS エラールーチンによって処理されます。このルーチンによってディスクアクセスの再試行が行われ、ユーザーはこの動作を中止するか、再試行するか、またはエラーを無視して続行するかを選択することができます。次に、割り込みタイプ 24H ルーチンに必要な条件、エラーコード、レジスタとスタックの管理について解説します。

エントリのステータス

MS–DOS は、I/O エラーに対して 3 回再試行した後、割り込みタイプ 24H を実行し、割り込み処理ルーチンは、割り込みタイプ 24H から制御を渡されます。AX と DI レジスタには、エラーについての情報が入ります。BP には、エラーを起こしたデバイスについて記述されているデバイスヘッダコントロールブロックのオフセットが入ります（セグメントアドレスは、SI に入ります）。

割り込みタイプ 24H ハンドラの必要条件

プログラムに「中止するか、再試行するか、無視するか」の選択をさせるプロンプトを表示する、MS–DOS の割り込みタイプ 24H の処理ルーチンを使いたい場合、ユーザーのエラー処理ルーチンは、フラグをプッシュし、標準的な割り込みタイプ 24H ハンドラのアドレスを FAR コールします（割り込みタイプ 24H のベクタを変更したユーザーのプログラムは、そのベクタアドレスをセーブしておかなくてはなりません）。ユーザーが前述のプロンプトに答えると、MS–DOS はユーザーのプログラムに制御を戻します。

ユーザーの割り込みハンドラでは、他の処理を行う前に、BX、CX、DX、EX、DS、SS、SP の内容を保存しなくてはなりません。使えるファンクションコールは 01H〜0CH、59H だけです（もし、他のファンクションコールを使うと、MS–DOS のスタック領域がこわされ、その後の動作は保証されません）。また、デバイスヘッダの内容を変えてはいけません。

注意　ユーザーが作成したアプリケーションで、スタック領域がこわされるようであれば、スタックフレームを変更してみるのもよいでしょう。

もし、割り込みタイプ 24H ルーチンから MS–DOS に戻らずにユーザーのプログラムに戻るときは、アプリケーションのレジスタをリストアし、スタックの最後の 3 ワードだけを残して IRET を行うと、エラーが生じた I/O ファンクションリクエストから直ちにプログラムに戻ります。この処理を行うと、MS–DOS は 0CH 以上のファンクションコールが行われるまで、不安定な状態となります。

エラーコード

・AX が返すディスクエラーコード

AH のビット 7 の 0 は、ディスクドライブに関連するエラーであることを示します。AL はエラーを起こしたドライブの番号（A：= 00H、B：= 01H、…）です。AH のビット 0 は、エラーの発生が、書き込み時か、読み込み時かを示します（0 ＝読み込み時、1 ＝書き込み時）。AH のビット 1 と 2 は、エラーを起こしたディスク領域の種類を示します。次に、その内容を示します。

ビット 2–1	種　類
00	MS–DOS 領域
01	ファイルアロケーションテーブル（FAT）
10	ディレクトリ
11	データ領域

AH のビット 3〜5 は、エラープロンプトに対する有効な返答を指定します。次に、その内容を示します。

ビット	内容	返　答
3	0	プログラムの失敗不可
	1	プログラムの失敗可
4	0	再試行不可
	1	再試行可
5	0	エラーの無視不可
	1	エラーの無視可

再試行が不可の場合、MS–DOS は再試行をせずに失敗したとみなします。エラーの無視を不可にすると、MS–DOS はエラーを無視せずに失敗したとみなします。プログラムの失敗を不可にすると、MS–DOS はプログラムを中止します。プログラムの中止は、常に可になっています。

・AX が返す他のデバイスのエラーコード

AH のビット 7 が 1 であると、ファイルアロケーションテーブル（FAT）のメモリイメージが悪いか、キャラクタデバイスにエラーがあることを示します。BP：SI によって指定されるデバイスヘッダには、デバイスとエラーの種類と属性を表す 1 ワードが含まれています。

属性を表す 1 ワードは、デバイスヘッダのオフセット 04H にあります。ビット 15 はデバイスの種類を示します（0 ＝ブロックデバイス、1 ＝キャラクタデバイス）。

ビット 15 が 0（ブロックデバイス）の場合、FAT のメモリイメージにエラーの原因があります。

ビット15が1（キャラクタデバイス）の場合、キャラクタデバイスにエラーの原因があります。DIがエラーコードを示し、ALのビット0〜3は、エラーを起こしたキャラクタデバイスの種類を示します。次に、その内容を示します。

ビット	内　容
0	標準入力
1	標準出力
2	NULデバイス
3	クロックデバイス

デバイスヘッダコントロールブロックの詳細については、「MS–DOSプログラマーズリファレンスマニュアルVol.2」を参照してください。

・DIが返すエラーコード

DIの下位バイトはエラーコードを示します。その内容を次に示します。上位バイトは不定です。

エラーコード	意　味
00H	ライトプロテクトされているディスクに書き込もうとした
01H	ドライブまたはユニットが存在しない
02H	ドライブの準備ができていない
03H	コマンドが無効
04H	データのCRCエラー
05H	リクエスト構造の長さが不正
06H	シークエラー
07H	メディアタイプが不正
08H	セクタが見つからない
09H	プリンタの用紙切れ
0AH	書き込みエラー
0BH	読み込みエラー
0CH	一般的なエラー

ユーザーの用意した割り込みタイプ24Hハンドラは、ファンクション59H（拡張エラーコードを得る）を実行すると、詳細なエラーの情報を得ることができます。

スタックの内容は次のとおりです。

スタックの一番上→ IP　INT 24Hが出た時点でのMS–DOSのレジスタ（致命的エラーによる割り込み）

CS
FLAGS
AX
BX
CX　　INT 21Hが出た時点でのユーザーレジスタ

DX
SI
DI
BP
DS
ES
IP
CS　　ユーザーから DOS への割り込み
FLAGS

再試行（リトライ）

レジスタには、動作の再試行を行うために必要とされるデータが入っています。AL に次の値の 1 つを入れ、IRET を実行して動作の指定をします。

値	動　作
00H	エラーを無視する
01H	再試行
02H	プログラムを打ち切る
03H	プログラム上のシステムコールの失敗

エラーを無視するオプションを指定すると、MS-DOS はエラーが生じていないと判断して処理を続けるため、予期せぬ状態になることが考えられますので注意してください。

再試行を指定する場合は、レジスタの内容を変更しないでください。

割り込みタイプ

25H　アブソリュートディスクリード

コール

AL　　=ドライブ番号（00H = A:、01H = B:、…）
DS:BX=ディスク転送アドレス（DTA）
CX　　=読み込みセクタ数
DX　　=読み込み開始相対セクタ番号

リターン

キャリーフラグがセットされる
　エラー発生　　ALにエラーコードを返す

キャリーフラグがリセットされる
　処理の正常終了

解　　説

指定したセクタからデータをメモリの指定した領域に読み込みます。
レジスタの値は以下のようになります。

AL ＝ ドライブ番号（A: = 00H、B: = 01H、…）
BX ＝ ディスク転送アドレスのオフセット（DS内のセグメントアドレスから）
CX ＝ 読み込むセクタ数
DX ＝ 読み込み開始相対セクタ番号

警　告

このシステムコールは、できるだけ使わないでください。ファイルのアクセスは、通常のファンクションコールで行ってください。というのは、このシステムコールでは、MS-DOSの上位バージョンに対する互換性が保証されないからです。この割り込みによって、制御は直接MS-DOSのデバイスドライバに移行します。CXで指定した数のセクタが、ディスクからディスク転送アドレスに読み込まれます。この割り込みの使い方や処理は、データが書き出されるのではなく読み込まれるということ以外は、割り込みタイプ26Hと同一です。なお、この割り込みは、使い方を誤ると動作が不安定になるので注意してください。

注意　セグメントレジスタ以外のすべてのレジスタの内容は、このコールによって破壊されるため、割り込みをかける前に、ユーザープログラムが使用するすべてのレジスタの内容を必ず保存してください。

このコールを行うとき、フラグはスタックに積まれ（プッシュフラグ：PUSHF)、システムによって終了した後も残ります（処理の結果を表すデータがフラグ内に返されるため)。スタックが制限なく増加することを防ぐために、終了後、必ずそのスタックを取り出してください（ポップフラグ：POPF)。

ディスク動作が正しく行われるとキャリーフラグ（CF）＝0に、正しく行われないとCF＝1になり、ALにエラーコードが返されます（エラーコードとその意味については、割り込みタイプ24Hを参照してください。)

マクロ定義

```
abs_disk_read   macro   disk, buffer, num_sectors, start
                mov     al, disk
                mov     bx, offset buffer
                mov     cx, num_sectors
                mov     dx, start
                int     25H
                popf
                endm
```

サンプル

次のプログラムは、ドライブAのディスクの内容を、ドライブBのディスクにコピーするものです。このプログラムでは、32Kバイトの大きさのバッファを使用しています。

```
prompt   db      "Source is A,  Destination is B", 13, 10
         db      "Any key to start.$"
start    dw      0
buffer   db      64 dup(512 dup(?))  ;64  sectors
;

int_25H: display prompt        ;prompt の内容を表示 (09H)
         read_kbd              ; キーボード入力待ち (08H)
         mov     cx, 5         ;1 回 (64 セクタ) の読み込み回数 (5) を設定
copy:    push    cx                ; 読み込みカウンタ (回数) をセーブ
         abs_disk_read   0, buffer, 64, start
                                   ; アブソリュートディスクリード
         abs_disk_write  1, buffer, 64, start
                                   ; アブソリュートディスクライト
         add     start, 64         ; 次の 64 セクタについて行う
         pop     cx                ; 読み込みカウンタをリストア
         loop    copy
```

割り込みタイプ

26H アブソリュートディスクライト

コール

AL　　=ドライブ番号
DS：BX=ディスク転送アドレス（DTA）
CX　　=書き出しセクタ数
DX　　=書き出し開始相対セクタ番号

リターン

キャリーフラグがセットされる
　エラー発生　　AL にエラーコードを返す

キャリーフラグがリセットされる
　処理の正常終了

解　　　説

ディスクの指定したセクタにメモリの内容を書き込みます。
レジスタの値は以下のようになります。

AL ＝ ドライブ番号（A：＝00H、B：＝01H, …）
BX ＝ ディスク転送アドレスのオフセット（DS 内のセグメントアドレスから）
CX ＝ 読み込むセクタ数
DX ＝ 読み込み開始相対セクタ番号

警　　告

メモリの内容をディスクに書き込むという違いだけで、割り込みタイプ 25H と同一です。したがって、25H と同じ理由で、このシステムコールはできるだけ使わないでください。また、スタックの処理なども割り込みタイプ 25H と同様、十分に注意してください。

マクロ定義

```
abs_disk_write  macro   disk, buffer, num_sectors, start
                mov     al, disk
                mov     bx, offset buffer
                mov     cx, num_sectors
                mov     dx, start
                int     26H
                popf
```

```
            endm
```

サンプル

次のプログラムは、ドライブAのディスクの内容を、ドライブBのディスクにコピーし、書き込み（ライト）が行われるごとに、ベリファイ（検証）を行うものです。このプログラムでは、32Kバイトの大きさのバッファを使用しています。

```
off         equ     0
off         equ     1
prompt      db      "Source in A,  Destination in B", 13, 10
            db      "Any key to start.$"
start       dw      0
buffer      db      64 dup(512 dup(?)) ;64 sectors
;
int_26H:    display prompt       ;prompt の内容を表示（09H)
            read_kbd             ; キーボード入力待ち（08H)
            verify  on           ; ベリファイフラグを ON にする（2EH)
            mov     cx, 5        ;1 回（64 セクタ）読み込み回数（5）を設定
copy:       push    cx           ; 書き出しカウンタ（回数）をセーブ
            abs_disk_read 0, buffer, 64, start  ; アブソリュート
                                                ; ディスクリード（25H)
            abs_disk_write 1, buffer, 64, start ; アブソリュート
                                                ; ディスクライト
            add     start, 64        ; 次の 64 セクタについて行う
            pop     cx               ; 書き出しカウンタをリストア
            loop    copy
            verify  off              ; ベリファイフラグを off にする（2EH)
```

割り込みタイプ
27H プロセスの常駐終了

コール　CS：DX＝コードの最終バイト＋1のアドレス

リターン　なし

解　説

プログラムサイズが64Kバイト以下のプロセスをメモリに常駐したまま終了させます。このコールは、デバイススペシフィック割り込みハンドラでよく使用されます。

この割り込みは、バージョン2.0以前のMS–DOSと互換性を保つために用意されています。バージョン2.0以降のMS–DOSを対象とするプログラムは、ファンクション31H（プロセスの常駐終了）を使用してください。このファンクションは、64Kバイトを超えるプログラムでもメモリに常駐させることができます。ユーザーが作ったプログラムが、バージョン2.0以前のMS–DOSに対する互換性を要求されない限り、常駐するプログラムにリターン情報を渡すことができます。

DXには、常駐させるプログラムコードの最終バイトの次に来る先頭のオフセット（CSのセグメントアドレスからの）が入っていなければなりません。割り込みタイプ27Hが実行されるとプログラムは終了し、制御はMS–DOSに戻ります。しかし、他のプログラムによるオーバーレイは行われません。ファイルはオープンされたままで、クローズされていません。割り込みが実行されたとき、CSには必ずPSP（割り込みが実行されたときのESとDSの値）のセグメントアドレスが入っていなければなりません。

この割り込みは、EXE形式のプログラムで使用することはできません。またこの割り込みは、割り込みタイプ22H、23H、24Hのベクタを保存しますので、新しい<CTRL–C>や致命的エラーのエラーハンドラを作ることができません。

マクロ定義

```
stay_resident   macro   last_instruc
        mov     dx, offset  last_instruc
        inc     dx
        int     27H
        endm
```

サンプル

```
;CSのセグメントアドレスは割り込み実行時のPSP値（ESとDSの値）と
;同じでなければならない
        mov     DX, LastAddress
        int     27H         ;この割り込みにはリターン情報はない
```

1.11 ファンクションリクエスト

次の表は、ファンクションリクエスト 00H〜68H についての解説です。

番号	ファンクションリクエスト
00H	プログラムの終了
01H	文字入力（エコーあり）
02H	文字出力
03H	補助入力
04H	補助出力
05H	文字のプリンタ出力
06H	直接コンソール入出力
07H	直接コンソール文字入力
08H	キーボード入力（エコーなし）
09H	文字列の表示
0AH	バッファードキーボード入力
0BH	キーボードステータスの検査
0CH	バッファを空にしてキーボード入力
0DH	ディスクのリセット
0EH	ディスクの選択
0FH	ファイルのオープン
10H	ファイルのクローズ
11H	最初のエントリを検索
12H	次のエントリを検索
13H	ファイルの削除
14H	シーケンシャルな読み出し
15H	シーケンシャルな書き込み
16H	ファイルの作成
17H	ファイル名の変更
19H	カレントドライブ番号の取得
1AH	ディスク転送アドレスの設定
1BH	デフォルトドライブのデータの取得
1CH	ドライブのデータの取得
21H	ランダムな読み出し
22H	ランダムな書き込み
23H	ファイルの大きさの取得
24H	相対レコードの設定
25H	割り込みベクタの設定
26H	新しい PSP の作成

番号	ファンクションリクエスト
27H	ランダムなブロックの読み出し
28H	ランダムなブロックの書き込み
29H	ファイル名の解析
2AH	日付の取得
2BH	日付の設定
2CH	時刻の取得
2DH	時刻の設定
2EH	ベリファイフラグのセット／リセット
2FH	ディスク転送アドレスの取得
30H	MS–DOS バージョン番号の取得
31H	プロセスの常駐終了
33H	<CTRL–C>チェックのセット／リセット
35H	割り込みベクタの取得
36H	ディスクのフリースペースの取得
38H	国別情報の取得
38H	国別情報の設定
39H	ディレクトリの作成
3AH	ディレクトリの削除
3BH	カレントディレクトリの変更
3CH	ハンドルを使うファイルの作成
3DH	ハンドルを使うファイルのオープン
3EH	ハンドルを使うファイルのクローズ
3FH	ファイルかデバイスの読み出し
40H	ファイルかデバイスへの書き込み
41H	ディレクトリエントリの削除
42H	ファイルポインタの移動
43H	ファイルの属性の取得／設定
4400H	IOCTL データの取得
4401H	IOCTL データの設定
4402H	IOCTL キャラクタを受け取る
4403H	IOCTL キャラクタを送る
4404H	IOCTL ブロックを受け取る
4405H	IOCTL ブロックを送る
4406H	入力ステータスのチェック
4407H	出力ステータスのチェック
4408H	媒体が交換可能か調べる
4409H	リモートブロックデバイスの検出
440AH	リモートハンドルの検出
440BH	リトライ回数の変更

番号	ファンクションリクエスト
440CH	一般 IOCTL（ハンドル用）
440DH	一般 IOCTL（ブロックデバイス用）
440EH	論理ドライブマップの取得
440FH	論理ドライブマップの設定
45H	ファイルハンドルの二重化
46H	ファイルハンドルの強制二重化
47H	カレントディレクトリの取得
48H	メモリの割り当て
49H	割り当てられたメモリの開放
4AH	割り当てられたメモリブロックの変更
4B00H	プログラムのロードと実行
4B03H	オーバーレイのロード
4CH	プロセスの終了
4DH	子プロセスからリターンコードを取得
4EH	最初に一致するファイル名の検索
4FH	次に一致するファイル名の検索
54H	ベリファイのステータスの取得
56H	ディレクトリエントリの変更
57H	ファイルの日付／時刻の取得／設定
58H	アロケーションストラテジの取得／設定
59H	拡張エラーコードの取得
5AH	一時ファイルの作成
5BH	新しいファイルの作成
5C00H	ファイルアクセスのロック
5C01H	ファイルアクセスのロック解除
5E00H	マシン名の取得
5E02H	プリンタセットアップ
5F02H	割り当てリストのエントリの取得
5F03H	割り当てリストのエントリの作成
5F04H	割り当てリストのエントリの取り消し
62H	PSP アドレスの取得

INT 21H

ファンクション 00H プログラムの終了

コール AH = 00H
CS = PSP のセグメントアドレス

リターン なし

解　説

ファンクション 00H は割り込みタイプ 20H（プログラムの終了）をコールし、同じ処理を行います。

この割り込みを実行するには、その前に CS レジスタへ PSP のセグメントアドレスを入れておかなければなりません。詳しくは割り込みタイプ 20H を参照してください。

PSP 内のオフセットに設定されていた抜け出しアドレスは、以下のように MS-DOS に戻されます。抜け出しアドレスについての詳細は、割り込みタイプ 22H から 24H の解説を参照してください。

抜け出しアドレス	オフセット	割り込みベクタ
プログラムの終了	0AH	INT 22H
<CTRL-C>	0EH	INT 23H
重大なエラー	12H	INT 24H

ファイルバッファの内容は、すべてディスクに書き出されます。

警　告

このファンクションコールを行うには、その前に大きさを変更したすべてのファイルをクローズしておかなければなりません。

変更されたファイルが先にクローズされていないと、ファイルは変更前のサイズでクローズされるので、正しく記録されません。ファイルのクローズについては、ファンクション 10H を参照してください。

マクロ定義

```
terminate_program   macro
                    xor     ah, ah
                    int     21H
                    endm
```

サンプル

メッセージを出力して、MS-DOSに戻るプログラムを示します。これは1.9「システムコールの使い方」のサンプルプログラムのようなプログラムのサブルーチンとして使われます。

```
message     db    "Displayed by FUNC_00H example", 0DH, 0AH, "$"
;
func_00H:   display message           ;messageを表示 (09H)
            terminate_program         ; プログラムの終了
code        ends
            end      start
```

INT 21H

ファンクション **01H**

文字入力（エコーあり）

コール　AH = 01H

リターン　AL =入力された文字

解　　説

標準入力（キーボード）から１文字入力されるまで待ち、入力された文字を標準出力（画面）に出力し、そのキャラクタコードを AL レジスタに返します。<CTRL–C>が入力されると、割り込みタイプ 23H を実行します。

マクロ定義

```
read_kbd_and_echo   macro
                    mov     ah, 01H
                    int     21H
                    endm
```

サンプル

次のプログラムは、文字を入力したとおりに画面とプリンタに出力します。リターンキーが押されると、改行コード（キャリッジリターンコード）が画面とプリンタの両方に出力されます。

```
func_01H:   read_kbd_and_echo               ; 文字入力（エコーあり）
            print_char      al              ; プリンタに出力（05H）
            cmp             al, 0DH         ; キャリッジリターンコードか?
            jne             func_01H        ; いいえのとき、次の文字の入力待ち
            print_char      10              ; 改行コードをプリンタに出力（05H）
            display_char    10              ; 改行コードを画面に出力（02H）
            jmp             func_01H        ; ループの先頭にジャンプ
```

INT 21H

ファンクション 02H 文字出力

コール　AH = 02H
DL =出力すべき文字コード

リターン　なし

解　説

DL 内の文字を標準出力に出力します。<CTRL–C>が入力されると、割り込みタイプ 23H が実行されます。

マクロ定義

```
display_char    macro   character
                mov     dl, character
                mov     ah, 02H
                int     21H
                endm
```

サンプル　次のプログラムは、小文字を大文字に変換して画面に表示します。

```
func_02H:       read_kbd                ; キーボード入力 (08H)
                cmp     al, "a"
                jl      uppercase       ; 変換しない (英小文字でない)
                cmp     al, "z"
                jg      uppercase       ; 変換しない (英小文字でない)
                sub     al, 20H         ; 大文字の ASCII コードに変換
uppercase:      display_char al         ; 大文字を画面に出力
                jmp     func_02H        ; 他の文字の入力待ち
```

INT 21H

ファンクション
03H 補助入力

コール AH ＝ 03H

リターン AL ＝補助装置から入力された文字

解　説

補助装置（AUX）から1文字入力されるまで待ち、入力された文字コードをALに返します。このファンクションコールは、ステータスやエラーコードを返しません。<CTRL–C>が入力されると、割り込みタイプ23Hが実行されます。

マクロ定義

```
aux_input       macro
                mov     ah, 03H
                int     21H
                endm
```

サンプル

次のプログラムは、補助装置から入力された文字を、そのままプリンタ出力します。エンドオブファイルコード（ASCIIコード1AH、<CTRL–Z>）が入力されると、出力を停止します。

```
func_03H:       aux_input               ；補助入力装置からの入力
                cmp     al, 1AH         ；ファイルエンドか？
                je      program_end     ；はいのとき、出力を停止
                print_char al           ；入力文字をプリンタに出力（05H）
                jmp      func_03H       ；他の文字の入力待ち
program_end     .
                .
                .
```

INT 21H

ファンクション 04H 補助出力

コール

AH = 04H
DL =補助装置に出力すべき文字

リターン

なし

解説

DL内の文字を、補助装置（AUX）に出力します。このファンクションコールは、ステータスやエラーコードを返しません。<CTRL–C>が入力されると、割り込みタイプ23Hが実行されます。

マクロ定義

```
aux_output      macro   character
                mov     dl, character
                mov     ah, 04H
                int     21H
                endm
```

サンプル

次のプログラムは、キーボードから入力された最高80バイトまでの一連の文字列を、補助装置に出力します。このプログラムは、ヌル文字列（CRのみ）が入力されると停止します。

```
string          db      81      dup(?)      ; ファンクション 0AH 参照
;
func_04H:       get_string 80, string       ; キーボードから最大 80 バイト
                                            ; 入力する (0AH)
                cmp     string[1], 0        ; ヌル文字列か?
                je      next_process        ; はいのとき、停止
                mov     cx, word ptr string[1]   ; 文字列長を得る
                mov     bx, 0               ; インデックス (BX) に 0 を設定
send_it:        aux_output string[bx+2]     ; 補助装置に出力
                inc     bx                  ; インデックスをインクリメント
                loop    send_it             ; 次の文字出力処理
```

```
                jmp     func_04H            ; 他の文字列の入力/出力処理へ
next_process:   .
                .
                .
```

INT 21H

ファンクション 05H　文字のプリンタ出力

コール　AH ＝05H
DL ＝プリンタに出力すべき文字

リターン　なし

解説

DL内の文字をプリンタ（PRN）に出力します。このファンクションはステータスやエラーコードを返しません。<CTRL–C>が入力されると、割り込みタイプ23Hが実行されます。

マクロ定義

```
print_char      macro   character
                mov     dl, character
                mov     ah, 05H
                int     21H
                endm
```

サンプル　次のプログラムは、プリンタ内にテストパターンを出力します。このプログラムは、<CTRL–C>が押されると停止します。

```
line_num      db      0
;
func_05H:     mov     cx, 60          ; プリンタ出力ライン数を 60 とする
start_line:   mov     bl, 33          ; 最初にプリント可能な ASCII
                                      ; 文字は（!）である
              add     bl, line_num    ; 出力する文字のオフセットを設定
              push    cx              ; プリンタ出力ラインカウンタをセーブ
              mov     cx, 80          ;1 行文の文字数（80）を CX に設定
print_it:     print_char bl           ; プリンタに文字を出力
              inc     bl              ; 次の ASCII 文字の出力準備
              cmp     bl, 126         ; 出力可能な最後の
                                      ;ASCII 文字（~）か?
              jl      no_reset        ; まだのときは no_reset へ
```

```
            mov         bl, 33          ; 文字 (!) から始める
no_reset:   loop        print_it        ; 次の文字の出力処理へ
            print_char  13              ; キャリッジリターンコードの出力
            print_char  10              ; ラインフィードコードの出力
            inc         line_num        ; オフセットをインクリメント
            pop         cx              ; プリンタ出力ラインカウンタをリストア
            loop        start_line      ; 次のラインをプリント
```

INT 21H

ファンクション 06H 直接コンソール入出力

コール

AH ＝06H
DL ＝解説の項を参照

リターン

AL
コールする前に、DL ＝FFHの場合：
ゼロフラグがセットされていなければALにキャラクタが入り、ゼロフラグがセットされていればキャラクタはなく、AL＝00Hになる。

コールする前に、DL ≠ FFHの場合：　なし

解　説

この処理は、ファンクションコールを行うときのDLによって次のように変わります。

DL ＝FFH
標準入力から文字が入力された場合、その文字をALに返し、ゼロフラグを0にします。文字が入力されていない場合、ゼロフラグを1にします。

DL ≠FFH
DL内の文字を、標準出力（画面）に出力します。

この機能は、<CTRL-C>の検査を行いません。

マクロ定義

```
dir_console_io  macro   switch
                mov     dl, switch
                mov     ah, 06H
                int     21H
                endm
```

サンプル

次のプログラムは、システムクロックを0に設定し、時刻を継続的に画面に表示します。なんらかの文字が入力されると、時刻の表示が停止します。再び文字が入力されると、このクロックは0にリセットされ、時刻の表示が再開します。

```
time            db"00:00:00.00", 13, 10, "$"   ;"$"の説明は
                                               ; ファンクション 09H 参照
ten               db      10
;
func_06H:       set_time        0, 0, 0, 0     ; 時刻を設定 (2DH)
read_clock:     get_time                       ; 時刻を得る (2CH)
                convert         ch, ten, time    ; 章末参照
                convert         cl, ten, time[3] ; 章末参照
                convert         dh, ten, time[6] ; 章末参照
                convert         dl, ten, time[9] ; 章末参照
                display         time           ;time を画面に表示 (09H)
                dir_console_io  0FFH    ; 任意の文字を入力
                jne             stop    ; 入力ありのとき、時刻表示の停止
                jmp             read_clock ; 入力なしのとき、
                                           ; 時刻表示の継続
                                           ;running
stop:           read_kbd                   ; キーボード入力待ち (08H)
                jmp             func_06H   ; 時刻表示を再開
```

INT 21H

ファンクション 07H　直接コンソール文字入力

コール　AH = 07H

リターン　AL =キーボードから入力された文字

解　　説

標準入力から文字が入力されるまで待ち、入力された文字を AL に返します。このファンクションは、文字のエコーや<CTRL-C>の検査は行いません。エコーまたは<CTRL-C>の検査を行うファンクションについては、ファンクション 01H または 08H を参照してください。

マクロ定義

```
dir_console_input       macro
                        mov     ah, 07H
                        int     21H
                        endm
```

サンプル

次のプログラムは、8文字までのパスワード入力を促すプロンプトを表示し、入力をエコーせずに、この文字を文字列内に入れます。

```
password      db       8 dup(?)
prompt        db       "password:$"         ;"$"の説明は
                                            ;ファンクション 09H 参照
func_07H:     display  prompt               ;prompt を画面に表示 (09H)
              mov      cx, 8                ;入力可能なパスワードの
                                            ;最大値 8 を設定
              xor      bx, bx               ;bx はパスワードのインデックス
                                            ;として使用
get_pass:     dir_console_input             ;キーボードから入力された文字を
                                            ;AL に返す
              cmp      al, 0DH              ;キャリッジリターンか?
              je       continue             ;はいのとき、処理終了
              mov      password[bx], al     ;いいえのとき、この文字を
                                            ;文字列内に入れる
```

INT 21H

ファンクション 08H 文字入力（エコーなし）

コール　AH ＝ 08H

リターン　AL ＝キーボードから入力された文字

解　説

標準入力（キーボード）から1文字入力されるまで待ち、この文字をALに返します。<CTRL-C>が入力されると、割り込みタイプ23Hが実行されます。このファンクションは、文字のエコーを行いません（文字のエコーを行う文字入力ファンクションについては、ファンクション01Hを参照してください）。

マクロ定義

```
read-kbd        macro
                mov     ah, 08H
                int     21H
                endm
```

サンプル

次のプログラムは、8文字までのパスワードの入力を促すためプロンプトを表示し、エコーを行わずに文字を文字列内に入れます。

```
password       db 8 dup(?)
prompt         db "password:$"          ;"$" の説明は
                                        ; ファンクション 09H 参照
;
func_08H:      display prompt           ;prompt を画面に表示 (09H)
               mov     cx, 8            ; 入力可能なパスワードの最大値 8 を設定
               xor     bx, bx           ;BX はパスワードのインデックスとして
                                        ; 使用
get_pass:      read_kbd                 ; キーボードから入力された文字を
                                        ;AL に返す
               cmp     al, 0DH          ; キャリッジリターンか?
               je      next_process     ; はいのとき、処理終了
               mov     password[bx], al ; いいえのとき、この文字を
```

INT 21H

ファンクション 09H　文字列の表示

コール

AH　　＝09H
DS：DX＝画面に出力する文字列の先頭アドレス

リターン

なし

解　説

DS：DXで先頭アドレスを指定した領域に格納されている文字列を、〝$〟が検出されるまで、標準出力に出力します。文字列の終わりは、〝$〟で指定してください。なお、$は出力されません。

マクロ定義

```
display         macro   string
                mov     dx, offset string
                mov     ah, 09H
                int     21H
                endm
```

サンプル

次のプログラムは、入力されたキーの16進コードを画面に出力します。

```
table   db      "0123456789ABCDEF"
sixteen db      16
result  db      "-00H", 13, 10, "$"         ;"$"の説明は
                                            ; 本文参照
func_09H:       read_kbd_and_echo           ; キーボード入力された文字を
                                            ; 画面に表示 (01H)
                convert al, sixteen, result[3]  ; 章末参照
                display result              ; 入力されたキーの16進コードを
                                            ; 画面に出力
                jmp     func_09H            ; 処理継続
```

INT 21H

ファンクション
0AH バッファードキーボード入力

コール AH ＝0AH
DS：DX＝入力バッファの先頭アドレス

リターン なし

解　　説

標準入力(キーボード)から入力された文字列を、以下のフォーマットで入力バッファに格納します。リターンが入力されると、ファンクションが終了します。入力した文字数が、(最大文字数−1)を越えると、それ以後に入力された文字は無視され、リターンキーを押すまで、ASCIIコードのBEL(07H)を標準出力に出力し続けます。

オフセット	内　容
1	CR（キャリッジリターンコード）を含むバッファ内の最大文字数（ユーザーが設定する）
2	実際に入力された、CRを含まない文字数（この値は、このファンクションによって設定される）
3～n	バッファ領域（オフセット1で指定した大きさ以上でなければならない）

入力時には、通常のコマンドラインと同様にテンプレートなど、各種の編集機能が使えます。
<CTRL–C>が入力されると、割り込みタイプ23Hが実行されます。
MS–DOSは、このバッファの2バイト目に、入力された文字数（CRを含まない）を設定します。

マクロ定義

```
get_string      macro   limit, string
                mov     dx, offset string
                mov     string, limit
                mov     ah, 0AH
                int     21H
                endm
```

サンプル

次のプログラムは、キーボードから最大16バイトまでの文字列を入力し、この文字列で24行×80文字の画面を埋めます。

```
buffer          label       byte
max_length      db          ?               ; 最大長
chars_entered   db          ?               ; 文字数
string          db          17dup(?)        ;16 文字＋リターンコード
string_per_line dw          0               ;1 行に出力可能な文字数
crlf            db          13, 10, "$"
;
func_0AH:       get_string  17, buffer ; バッファードキーボード入力
                xor         bx, bx     ;BX はバッファのインデックス
                                       ; としてバイト単位で使用
                mov         bl, chars_entered ; 文字列長を得る
                mov         buffer[bx+2], "$" ;"$" の設定 (09H)
                mov         al, 50H    ; ラインあたりのカラム数を指定
                cbw
                div         chars_entered ;1 行あたりの文字数を算出
                xor         ah, ah        ; 残りをクリア
                mov         strings_per_line, ax  ; カラムカウンタを
                                                  ; セーブ
                mov         cx, 24            ; ラインカウンタを設定
display_screen: push        cx                ; それをセーブ
                mov         cx, strings_per_line  ; カラムカウンタを得る
display_line:   display     string            ;string を画面に表示 (09H)
                loop        display_line
                display     crlf              ;CRLF を画面に出力 (09H)
                pop         cx                ; ラインカウンタを得る
                loop        display_screen    ; 次の 1 行表示へ
```

INT 21H

ファンクション 0BH　キーボードステータスの検査

コール　AH ＝ 0BH

リターン　AL ＝ FFH　タイプアヘッドバッファ内に文字が入っている
　　　　＝ 00H　タイプアヘッドバッファ内に文字が入っていない

解　説

標準入力（標準入力がリダイレクトでなければタイプアヘッドバッファ内）に、文字が入っているかどうかを検査します。入っているとALにFFH（255）が、入っていないと00Hが返されます。<CTRL-C>がバッファ内に入っていると、割り込みタイプ23Hが実行されます。

マクロ定義

```
check_kbd_status        macro
                        mov     ah, 0BH
                        int     21H
                        endm
```

サンプル

次のプログラムは、いずれかのキーが押されるまで、時刻を継続的に画面に出力します。

```
time      db        "00:00:00.00", 13, 10, "$"
ten       db        10
                    .
                    .
func_0BH:           get_time                          ; 時刻を得る (2CH)
                    convert         ch, ten, time     ; 章末参照
                    convert         cl, ten, time[3]  ; 章末参照
                    convert         dh, ten, time[6]  ; 章末参照
                    convert         dl, ten, time[9]  ; 章末参照
                    display         time              ; 時刻の表示 (09H)
                    check_kbd_status              ; キーボードステータスの検査
                    cmp       al, 0FFH            ; タイプアヘッドバッファ内に
                                                  ; 文字が入っているか?
```

```
                je          all_done          ; はいのとき、処理終了
                jmp         func_0BH          ; いいえのとき
                                              ; 時刻を継続的に
                                              ; 画面出力
all_done:       .
                .
```

INT 21H

バッファを空にしてキーボード入力

コール

AH ＝ 0CH
AL ＝ 01H、06H、07H、08H、0AH：
対応するファンクションのコールが行われる
他の値：
これ以上の処理は行われない

リターン

AL ＝ 00H
タイプアヘッドバッファは、空になっている。これ以上の処理は行われないファンクションを指定した場合は、そのファンクションのリターンに準じます。

解説

標準入力バッファ（標準入力がリダイレクトでなければタイプアヘッドバッファ）を空にします。これ以上の処理を行うかどうかは、このファンクションコールが行われたときの AL の値によります。

01H、06H、07H、08H、0AH……対応する MS-DOS ファンクションが、実行されます。
他の値……これ以上の処理は行われず、AL に 0 が返されます。

マクロ定義

```
flush_and_read_kbd      macro   switch
                        mov     al, switch
                        mov     ah, 0CH
                        int     21H
                        endm
```

サンプル

次のプログラムは、文字を入力したとおりに画面とプリンタに出力します。リターンキーが押されると、改行コード（キャリッジリターンコード）が画面とプリンタの両方に出力されます。

```
func_0CH:    flush_and_read_kbd 01H ；バッファを空にしてキーボード入力
                                    ；ファンクション 01H を指定
             print_char    al       ；入力された文字をプリンタに出力（05H）
             cmp           al, 0DH  ；キャリッジリターンか？
```

```
jne             func_0CH ; いいえのとき、プリンタに出力
print_char      10       ; はいのとき、プリンタに
                         ; 改行コードを出力（05H）
display_char    10       ; 画面に改行コードを出力（02H）
jmp             func_0CH ; 次の文字を得る
```

INT 21H

ディスクのリセット

コール　AH = 0DH

リターン　なし

解　　説

このファンクションは、すべてのファイルバッファの内容をディスクに書き出し、ファイルバッファを空にします。ディレクトリエントリの更新は行わないので、ユーザーはディレクトリエントリの更新を行うために変更されたファイルをクローズしなければなりません。ファイルのクローズについては、ファンクション 10H（ファイルのクローズ）やファンクション 3EH（ハンドルのクローズ）を参照してください。

マクロ定義

```
reset_disk      macro   disk
                mov     ah, 0DH
                int     21H
                endm
```

サンプル　次のプログラムは、ファイルバッファを空にします。

```
reset_disk      ; このコールにエラーリターンはない
```

INT 21H

ファンクション 0EH ディスクの選択

コール　AH = 0EH
DL =ドライブ番号（A：= 00H、B：= 01Hなど）

リターン　AL =論理ドライブの数

解　　説

DLで指定されたドライブ（00H = A：、01H = B：、…）が、カレントのディスクとして選択されます。ALにドライブ数が返されます。

注意　将来の互換性のために、ALに返された値は注意深く扱ってください。ALに返される論理ドライブ数は、実際に接続されているドライブ数とは限らず、CONFIG.SYSのLASTDRIVE指定によって変わります。ALの最小値は5（LASTDRIVE指定なしで実際のドライブが5台以下のとき）なので、ALが05Hを返しても、A、B、C、D、Eの各ドライブがすべて有効なドライブ指定とは限りません。

マクロ定義

```
select_disk     macro   disk
                mov     dl, disk-"A"
                mov     ah, 0EH
                int     21H
                endm
```

サンプル

次のプログラムは、2ドライブシステムで現在選択されていないドライブを、カレントディスクにします。

```
func_0EH:       current_disk            ; カレントドライブ番号を得る (19H)
                cmp     al, 00H         ; ドライブ A が選択されているか?
                je      select_b        ; はいのとき、select_b へ
                select_disk "A"         ; いいえのとき、ドライブ A を選択
                jmp    next_process
select_b:       select_disk "B"         ; ドライブ B を選択
next_process:   .
                .
```

INT 21H

ファンクション 0FH ファイルのオープン

コール

AH ＝0FH
DS：DX＝オープンされていないFCBの先頭アドレス

リターン

AL ＝00H ディレクトリエントリが存在する
＝FFH ディレクトリエントリが存在しない

解説

FCBで指定したファイルをオープンします。DXには、オープンされていないファイルコントロールブロック（FCB）のオフセット（DS内のセグメントアドレスから）が入っていなければなりません。指定された名前のファイルを見つけるために、ディスクディレクトリを検索します。

このファイルのディレクトリエントリが存在すると、ALに00Hが返され、FCBは次のように設定されます。

・ドライブコードが00H（カレントディスク）の場合、実際に使用されているディスク番号（01H＝A：、02H＝B：、…）に変更されます。このため、このファイルで引き続き行われる操作を妨害することなく、カレントディスクを変更することができます。

・現在のブロックフィールド（オフセット0CH）は、ゼロに設定されます。

・レコードサイズ（オフセット0FH）は、システムデフォルト値である128に設定されます。

・ファイルのサイズ（オフセット10H）、最後に書き込みが行われた日付（オフセット14H）と時刻（オフセット16H）が、ディレクトリエントリから得られた情報により設定されます。

このファイルに対してシーケンシャルなディスクアクセスを行うとき、事前に現在のレコードフィールド（オフセット20H）を、ランダムなディスクアクセスを行うときは、相対レコードフィールド（オフセット21H）を設定しておかなければなりません。デフォルトレコードサイズ（128バイト）を使用しないときは、正しい長さに設定してください。

ファイルのディレクトリエントリが存在しないか、属性がシステムあるいは隠されたファイルの場合、ALにFFH（255）が返されます。

このファンクションは、MS-DOSの古いバージョンとの互換性を保つために用意されているものです。プログラムを新規に作成する場合は、ファンクション3DH（ハンドルのオープン）をもちいてファ

イルをオープンするようにしてください。

マクロ定義

```
open    macro   fcb
        mov     dx, offset fcb
        mov     ah, 0FH
        int     21H
        endm
```

サンプル

次のプログラムは、ドライブBにあるディスク上のTEXTFILE.ASCという名前のファイルをプリンタに出力します。バッファ中のレコードに、エンドオブファイルコード（ASCIIコード1AH、<CTRL-Z>）が含まれていると、そのコードが検出されるまで文字が出力されます。

```
fcb          db          2, "TEXTFILE.ASC"
             db          25 dup(?)
buffer       db          128 dup(?)
;
func_0FH:    set_dta     buffer        ;ディスク転送アドレスの設定（1AH）
             open        fcb           ;TEXTFILE.ASCファイルのオープン
read_line:   read_seq    fcb           ;シーケンシャルな読み出し（14H）
             cmp         al, 1AH       ;ファイルエンドか?
             je          all_done      ;はいのとき、all_doneへ
             cmp         al, 00H       ;ディレクトエントリが存在するか?
             jg          check_more    ;いいえのとき、check_moreへ
                                       ;record
             mov         cx, 128       ;はいのとき、バッファ中のレコードを
                                       ;プリンタに出力
             xor         si, si        ;インデックスを0に設定
print_it:    print_char  buffer[si]    ;バッファ中の文字をプリンタ
                                       ;に出力（05H）
             inc         si            ;インデックスをインクリメント
             loop        print_it      ;次の文字をプリンタに出力
             jmp         read_line     ;次のレコードをリード
check_more:  cmp         al, 03H       ;プリントするレコードがあるか?
             jne         all_done      ;いいえのとき、all_doneへ
             mov         cx, 128       ;はいのとき、バッファ中のレコードを
                                       ;プリンタに出力
             xor         si, si        ;インデックスを0に設定
find_eof:    cmp         buffer[si], 1AH;ファイルエンドか?
             je          all_done        ;はいのとき、all_doneへ
             print_char  buffer[si]    ;バッファ中の文字をプリンタに出力
```

```
                                        ; (05H)
            inc         si              ; インデックスをインクリメントする
            loop        find_eof
all_done:   close       fcb             ; ファイルのクローズ (10H)
```

INT 21H

ファンクション 10H ファイルのクローズ

コール

AH　　＝10H
DS：DX＝オープンされている FCB

リターン

AL ＝00H　ディレクトリエントリが存在する
　　＝FFH　ディレクトリエントリが存在しない

解　　説

FCB をもちいてオープンされたファイルをクローズします。DX には、オープンされている FCB のオフセット（DS にはセグメントアドレス）が入っていなければなりません。FCB で指定されたファイルを見つけるために、ディスクディレクトリの検索が行われます。ファイルを変更したとき、このファンクションコールを行わないとディレクトリエントリは更新されません。

このファイルのディレクトリエントリが存在するとき、ファイルのロケーションが、FCB 内の対応するエントリと比較されます。必要に応じて FCB と一致させるため、エントリを更新し AL に 00H が返されます。

ファイルのディレクトリエントリが存在しないと、AL に FFH（255）が返されます。

このファンクションは、MS-DOS の古いバージョンとの互換性を保つために用意されているものです。プログラムを新規に作成する場合は、ファンクション 3EH（ハンドルのクローズ）をもちいてファイルをオープンするようにしてください。

マクロ定義

```
close   macro   fcb
        mov     dx, offset fcb
        mov     ah, 10H
        int     21H
        endm
```

サンプル

次のプログラムは、ドライブ B に存在する MOD1.BAS という名前のファイルの先頭のバイトが FFH かどうか調べ、FFH であるとプリンタにメッセージを出力します。

```
message     db      "Not saved in ASCII format", 13, 10, "$"
fcb         db      2, "MOD1    BAS"
```

```
            db        25  dup(?)
buffer      db        128 dup(?)
;
func_10H:   set_dta   buffer        ; ディスク転送アドレスの設定 (1AH)
            open      fcb           ;MOD1.BAS ファイルのオープン (0FH)
            read_seq  fcb           ; シーケンシャルな読み出し (14H)
            cmp       buffer, 0FFH  ; ファイルの先頭バイトは FFH か?
            jne       all_done      ; いいえのとき、all_done へ
            display   message       ; はいのとき、message を
                                    ; プリンタへ出力 (09H)
all_done:   close     fcb           ; ファイルのクローズ
```

INT 21H

ファンクション
11H 最初のエントリを検索

コール

AH　　=11H
DS：DX=オープンされていないFCB

リターン

AL =00H　ディレクトリエントリが存在する
　　=FFH　ディレクトリエントリが存在しない

解　説

カレントディレクトリを検索して、FCBのファイル名(ワイルドカードの使用も可)に一致するディレクトリエントリが存在すれば、そのディレクトリ情報をFCBと同じ形式でDTAにセットします。DXには、オープンされていないFCBのオフセット（DSにはセグメントアドレス）が入っていなければなりません。隠されたファイルやシステムファイルを検索する場合、DXは拡張FCBの先頭のバイトを示していなければなりません。

FCB内のファイル名のディレクトリエントリが存在する場合、ALに0が返され、同じ種類（通常または拡張）のオープンされていないFCBが、ディスク転送アドレスに作成されます。

FCB内のファイル名のディレクトリエントリが存在しない場合、ALにFFH（255）が返されます。

検索しているFCBが通常のFCBであると、ディスク転送アドレスの最初の1バイトには、使われているドライバ番号が設定され、次の32バイトがディレクトリエントリです。

検索しているFCBが拡張FCBであると、ディスク転送アドレスの最初の1バイトにはFFHが、次の5バイトには00Hが設定され、それに続く1バイトに検索しているファイルの属性が示されます。残りの33バイトは、通常のFCBのときと同じです（1バイトのドライブ番号と32バイトのディレクトリエントリ）。

ファンクション12H（次のエントリを検索）を使ってファイル名を検索する場合、DS：DXにある、元のFCBは、決してオープンしたり変更したりしないでください。

注意　属性フィールドは、拡張FCBの最後のバイトで、FCBの前に位置します（拡張FCBの詳細は、1.8「バージョン2.0以前のシステムコールの拡張FCB」についての解説を参照）。
拡張FCBが使われていると、次のような検索が行われます。

1. FCBの属性がゼロであると、通常のファイルエントリだけが検索される。ボリュームラベル、サブディレクトリ、隠されたシステムファイルは検索されない。
2. 属性フィールドが、隠れたファイル、システムファイル、ディレクトリエントリ（02H、04H、10H）またはその任意の組み合わせに設定されると、通常のファイルの他に、こ

れらのファイルも検索されるようになる。これは属性バイトが16H（隠された＋システム＋ディレクトリの3ビットすべてがON）に設定されたときで、ボリュームラベルだけは除外される。

3. 属性フィールドがボリュームラベル（08H）に設定されると、ボリュームラベルエントリだけが検索され、他は対象から除外される。

マクロ定義

```
search_first    macro   fcb
                mov     dx, offset fcb
                mov     ah, 11H
                int     21H
                endm
```

サンプル

次のプログラムは、ドライブBに〝REPORT.ASM〟という名前のファイルが存在するかどうか検索します。

```
yes     db      "FILE EXISTS.$"
no      db      "FILE DOES NOT EXIST.$"
fcb     db      2, "REPORT ASM"
        db      25 dup(?)
buffer  db      128 dup(?)
;
func_11H:       set_dta      buffer       ;ディスク転送アドレスの設定（1AH）
                search_first fcb          ;REPORT.ASMファイルの検索
                cmp          al, FFH      ;ディレクトリエントリが
                                          ;存在するか?
                je           not_there    ;いいえのとき、not_thereへ
                display      yes          ;はいのとき、yesを画面に出力（0AH）
                jmp          next_process
not_there:      display      no           ;noを画面に表示（09H）
next_process:   display      crlf         ;CRLFを画面に出力（09H）
```

INT 21H

ファンクション 12H 次のエントリを検索

コール

AH　　＝12H
DS：DX＝オープンされていないFCB

リターン

AL ＝00H　ディレクトリエントリが存在する
　 ＝FFH　ディレクトリエントリが存在しない

解説

ファンクション11H（最初のディレクトリエントリの検索）で名前の一致したディレクトリエントリから後の部分のディレクトリを対象にファイルを検索します。

DXには、前のファンクション11Hのコールのときに指定されたFCBのオフセット（DS内のセグメントアドレスから）が入っていなければなりません。このファンクションは、ファイル名にワイルドカード文字が使われたとき、他のディレクトリエントリを見つけるために、ファンクション11H（最初のエントリを検索）の後で使用します。ファイル名にはワイルドカード文字を使用することができます。隠れて見えないファイルまたはシステムファイルを検索する場合、DXは拡張FCBの先頭のバイトを示していなければなりません。

FCBにファイル名のディレクトリエントリが存在するとALに00Hが返され、同じ種類（通常または拡張）のオープンされていないFCBが、ディスク転送アドレスに作成されます。

FCBにファイル名のディレクトリエントリが存在しないと、ALにFFH（255）が返されます（オープンされていないFCBについてはファンクション11Hを参照してください）。

マクロ定義

```
search_next     macro   fcb
                mov     dx, offset fcb
                mov     ah, 12H
                int     21H
                endm
```

サンプル

次のプログラムは、ドライブBに存在するファイル数を画面に出力します。

```
message     db  "No files", 10, 13, "$"
files       db  0
ten         db  10
```

```
fcb          db  2, "???????????"
             db  25 dup(?)
buffer       db  128 dup(?)
;
func_12H:    set_dta   buffer          ; ディスク転送アドレスの設定（1AH）
             search_first fcb          ; 最初のエントリを検索（11H）
             cmp       al, 0FFH        ; ディレクトリエントリが存在するか?
             je        all_done        ; いいえのとき、all_done へ
             inc       files           ; はいのとき、ファイル数に 1 を加える
                                       ;counter
search_dir:  search_next fcb           ; 次のエントリを検索
             cmp       al, 0FFH        ; ディレクトリのエントリが存在するか?
             je        done            ; はいのとき、ファイル数に 1 を加える
             inc       files           ;counter
             jmp       search_dir      ; 再チェックする
done:        convert   files, ten, message      ; 章末参照
all_done:    display   message         ;message を画面に表示（09H）
```

INT 21H

ファンクション 13H ファイルの削除

コール

AH　　=13H
DS：DX=オープンされていないFCB

リターン

AL =00H　ディレクトリエントリが存在する
　 =FFH　ディレクトリエントリが存在しない

解　説

FCBで指定したファイルを削除します。

DXには、オープンされていないFCBのオフセット（DSにはセグメントアドレス）が入っていなければなりません。目的のファイル名を見つけるために、ディレクトリが検索されます。FCB内のファイル名には、ワイルドカード文字を使うことができます。

一致するディレクトリエントリが存在すると、このエントリはディレクトリから削除され、ALに00Hが返されます。このファイル名にワイルドカード文字が使用されていると、該当するすべてのディレクトリエントリが削除されます。

一致するディレクトリエントリが存在しないと、ALにFFH（255）が返されます。

マクロ定義

```
delete      macro   fcb
            mov     dx, offset fcb
            mov     ah, 13H
            int     21H
            endm
```

サンプル

次のプログラムは、ドライブBに存在するファイルのうち、1990年12月31日以前に編集されたものを削除します。

```
year      dw      1990
month     db      12
day       db      31
files     db      0
ten       db      10
message   db      "NO FILES DELETED.", 13, 10, "$"
```

```
                                        ;"$"の説明はファンクション 09H を参照
fcb         db      2, "???????????"
            db      25 dup(?)
buffer      db      128 dup(?)
;
func_13H: set_dta buffer              ; ディスク転送アドレスの設定 (1AH)
            search_first  fcb         ; 最初のエントリの検索 (11H)
            cmp           al, 0FFH    ; ディレクトリエントリは存在するか?
            je            all_done    ; いいえのとき、all_done へ
compare:    convert_date  buffer      ; 章末参照
            cmp           cx, year    ;CX (年) DL (月) DH (日) を
            jg            next        ; それぞれ year、month、day と
            cmp           dh, month   ; 比較する
            jg            next        ;1990 年 12 月 31 日以前ならば
            cmp           dh, day     ; ファイルを削除
            jge           next
            delete        buffer      ; ファイルの削除
            inc           files       ; 削除ファイルカウンタを
                                      ; インクリメントする
next:       search_next   fcb         ; 次のエントリを検索 (12H)
            cmp           al, 00H     ; ディレクトリエントリは存在するか?
            je            compare     ; はいのとき、日付のチェック
            cmp           files, 0    ; いくつかファイルを削除したか?
            je            all_done    ; いいえのとき NO FILES メッセージを表示
            convert       files, ten, message  ; 章末参照
all_done: display         message     ;message を画面に表示 (09H)
```

INT 21H

ファンクション 14H　シーケンシャルな読み出し

コール

AH　＝14H
DX：DX＝オープンされているFCB

リターン

AL　＝00H　正常な読み込み
　　＝01H　EOF
　　＝02H　ディスク転送アドレス（DTA）で示されるバッファが小さすぎる
　　＝03H　EOF、レコードの一部分

解　説

ファイルからFCBで示されるレコードサイズに等しいバイト数が、DTAで指定したバッファに読み出されます。

DXには、オープンされているFCBのオフセット（DSにはセグメントアドレス）が入っていなければなりません。カレントブロック（オフセット0CH）とカレントレコード（オフセット20H）フィールドが示すレコードが、ディスク転送アドレスにロードされ、次にカレントブロックとカレントレコードフィールドが、次のレコードを示すように設定されます。

レコードサイズフィールドは、FCB内のオフセット0EHにある値に設定されます。

ALに返されるコードは、次の処理が行われたことを示します。

コード	意　味
00H	リード（読み出し）が正しく行われ、処理完了した。
01H	ファイルの終わり。このレコードにデータは入っていない。
02H	ディスク転送アドレス内に、1レコードを読み出すのに十分な領域がなく、読み出しは取り消された。
03H	ファイルの終わり。<EOF>までのデータが読み出され、レコードの残りの部分がゼロで埋められた。

マクロ定義

```
read_seq    macro   fcb
            mov     dx, offset fcb
            mov     ah, 14H
            int     21H
            endm
```

サンプル

次のプログラムは、ドライブBのTEXTFILE.ASCという名前のファイルを画面に出力します。このファンクションは、MS–DOSのTYPEコマンドに似ています。読み出したレコードの途中にEOF（エンドオブファイル：ASCIIコード1AH、<CTRL–Z>）があると、EOFまでの文字が画面に出力されます。

```
fcb         db        2, "TEXTFILEASC"
            db        25 dup(?)
buffer      db        128 dup(?), "$"
;
func_14H:   set_dta   buffer             ;ディスク転送アドレスの設定（1AH）
            open      fcb                ;TEXTFILE.ASCファイルを
                                         ;オープン（0AH）
read_line:  read_seq  fcb                ;シーケンシャルな読み出し
            cmp       al, 02H            ;読み込みが取り消されたか?
            je        all_done           ;はいのとき、all_doneへ
            cmp       al, 00H            ;ファイルエンドか?
            jg        check_more         ;はいのとき、check_moreへ
            display   buffer             ;bufferを画面に表示（09H）
            jmp       read_line          ;次のレコードを得る
check_more: cmp       al, 03H            ;レコードの残りが読み込まれたか?
            jne       all_done           ;いいえのとき、all_doneへ
            xor       si, si             ;インデックスを0に設定
find_eof:   cmp       buffer[si], 1AH    ;EOFキャラクタか?
            je        all_done           ;はいのとき、all_doneへ
            display_char buffer[si]      ;バッファ中の文字を画面に出力（02H）
            inc       si                 ;インデックスをインクリメント
            jmp       find_eof           ;次の文字をチェック
all_done:   close     fcb                ;ファイルをクローズ（10H）
```

14H

INT 21H

ファンクション 15H シーケンシャルな書き込み

コール

AH　　＝15H
DS：DX＝オープンされている FCB

リターン

AL ＝00H　正常な書き込み
　　＝01H　ディスクに空き領域がない
　　＝02H　ディスク転送アドレス（DTA）で示されるバッファが小さすぎる

解　説

FCB で示されるレコードサイズに等しいバイト数が、DTA で指定したバッファからファイルに書き込まれます。

DX には、オープンされている FCB のオフセット（DS にはセグメントアドレス）が入っていなければなりません。カレントブロック（オフセット 0CH）とカレントレコード（オフセット 20H）フィールドが示すレコードにディスク転送アドレスから書き込まれ、次にカレントブロックとカレントレコードフィールドが、次のレコードを示すように設定されます。

レコードサイズは、FCB 内のオフセット 0EH にある値に設定されます。レコードサイズが 1 セクタよりも小さいと、ディスク転送アドレスにあるデータがバッファに移され、このバッファに入れられたデータが 1 セクタに達すると、ファイルのクローズか、ディスクのリセットシステムコール（ファンクション 0DH）の実行によって、このバッファがディスクに書き込まれます。

AL に返されるコードは、次の処理が行われたことを示します。

コード	意　味
00H	転送が正しく行われ、処理完了した。
01H	ディスクに空き領域がなく、書き込みは中止された。
02H	ディスク転送アドレスに、1 レコードを書き込むための十分な領域がないので、書き込みは中止された。

マクロ定義

```
write_seq   macro   fcb
            mov     dx, offset fcb
            mov     ah, 15H
            int     21H
            endm
```

サンプル

次のプログラムは、ドライブBにDIR.TMPという名前のファイルを作成します。このファイルには、ディスクの番号（A：＝01H、B：＝02H、…）とファイル名が入ります。

```
record_size     equ     14              ;FCBの中のレコードサイズフィールドの
                                        ;オフセット
;
fcb1            db      2, "DIR TMP"
                db      25 dup(?)
fcb2            db      2, "???????????"
                db      25 dup(?)
buffer          db      128 dup(?)
;
func_15H:       set_dta         buffer      ;ディスク転送アドレスの設定（1AH）
                search_first    fcb2        ;最初のエントリを検索
                cmp             al, 0FFH    ;ディレクトリエントリが
                                            ;存在するか?
                je              all_done    ;いいえのとき、all_doneへ
                create          fcb1        ;DIR.TMPファイルの作成（16H）
                mov             fcb1[record_size], 12
                                            ;レコードサイズ12を設定
write_it:       write_seq       fcb1        ;シーケンシャルな書き込み
                cmp             al, 0
                jne             all_done    ;正常終了ならall_doneへ
                search_next     fcb2        ;次のエントリを検索
                cmp             al, FFH     ;エントリが存在するか？
                je              all_done    ;いいえのとき、all_doneへ
                jmp             write_it    ;はいのとき、レコードをライト
all_done:       close           fcb1        ;ファイルをクローズ（10H）
```

INT 21H

ファンクション 16H ファイルの作成

コール

AH　　＝16H
DS：DX＝オープンされていないFCB

リターン

AL ＝00H　空のディレクトリエントリが存在する
　　＝FFH　空のディレクトリエントリが存在しない

解　説

ファイルを新規にオープンします。

DXには、オープンされていないFCBのオフセット（DSにはセグメントアドレス）が入っていなければなりません。空のエントリまたは指定されたファイル名の既存のエントリを見つけるために、ディレクトリが検索されます。

空のディレクトリエントリが存在すると、このエントリはファイルサイズゼロに初期値設定され、オープンファイルファンクション（0FH）が行われて、ALに00Hが返されます。属性バイト（オフセットFCB–1）を2に設定した拡張FCBを使用すると、隠されたファイルを作成することができます。

指定されたファイル名のエントリが存在すると、このファイル名に対してオープンファイルシステムコール（ファンクション0FH）が行われます。すなわち、既存のファイルは消去され、新規の空のファイルが作成されることになります。

空のディレクトリエントリも指定されたファイル名のエントリも存在しないと、ALにFFH（255）が返されます。

マクロ定義

```
create      macro   fcb
            mov     dx, offset fcb
            mov     ah, 16H
            int     21H
            endm
```

サンプル

次のプログラムは、ドライブBにDIR.TMPという名前のファイルを作成します。このファイルには、ディスクの番号（A：＝01H、B：＝02H、…）と、このディスク上のファイル名が入ります。

```
record_size     equ     14          ;FCBの中のレコードサイズフィールドの
```

```
                                        ; オフセット
;
fcb1            db      2, "DIR   TMP"
                db      25 dup(?)
fcb2            db      2, "???????????"
                db      25 dup(?)
buffer          db      128 dup(?)
;
func_16H:  set_dta       buffer     ; ディスク転送アドレスのセット (1AH)
           search_first fcb2       ; 最初のエントリを検索 (11H)
           cmp          al, 0FFH   ; ディレクトリエントリは存在するか?
           je           all_done   ; いいえのとき、all_done へ
           create       fcb1       ;DIR.TMP ファイルの作成
           mov          fcb1[record_size], 12
                                   ; レコードサイズ 12 を設定
write_it:  write_seq    fcb1       ; シーケンシャルな書き込み (15H)
           search_next  fcb2       ; 次のエントリを検索 (12H)
           cmp          al, 0FFH   ; ディレクトリエントリが存在するか?
           je           all_done   ; いいえのとき、all_done へ
           jmp          write_it   ; はいのとき、レコードをライト
all_done:  close        fcb1       ; ファイルをクローズ (10H)
```

16H

INT 21H

ファンクション 17H　ファイル名の変更

コール　AH　＝17H
DS：DX＝修正されたFCB

リターン　AL ＝00H　ディレクトリエントリが存在する
＝FFH　目的のディレクトリエントリが存在しないか、ファイル名がすでに存在している

解　説

既存するファイルを指定したファイル名に変更します。

DXには、FCBのオフセット（DSにはセグメントアドレス）が入っていなければなりません。このFCBにはドライブ番号とファイル名の後に、新しいファイル名がオフセット11Hから入っていなければなりません。修正したいファイル名（ワイルドカード文字を使うことができます）と一致するエントリを探すために、このディスクディレクトリが検索されます。

一致するディレクトリエントリが存在し、かつ2番目のファイル名が存在しないと、ディレクトリエントリのファイル名は、修正用FCBの新しいファイル名に変更されます（新旧のファイル名が、同じであってはなりません）。新しい2番目のファイル名にワイルドカード文字〝?″が使われていると、古いファイル名の対応する文字は変更されません。処理が完了すると、ALに00Hが返されます。

このファンクションは、隠しファイルまたはシステムファイルへ使用できません。一致するディレクトリエントリが存在しないか、2番目のファイル名のエントリがすでに存在すると、ALにFFH（255）が返されます。

マクロ定義

```
rename  macro   fcb
        mov     dx, offset fcb
        mov     ah, 17H
        int     21H
        endm
```

サンプル

次のプログラムは、変更したいファイル名と新しいファイル名の入力するプロンプトを出力し、ファイル名の変更を行います。

```
fcb     db      37 dup(?)
```

```
prompt1   db      "Filename: $"
prompt2   db      "New name: $"
reply     db      17 dup(?)
crlf      db      13, 10, "$"
;
func_17H: display prompt1              ;prompt1 を画面に表示 (09H)
          get_string 15, reply         ; バッファードキーボード入力 (0AH)
          display     crlf             ;crlf を画面に出力 (09H)
          parse       reply[2], fcb    ; ファイル名の解析 (29H)
          display     prompt2          ;prompt2 を画面に表示 (09H)
          get_string 15, reply         ; バッファードキーボード入力 (0AH)
          display     crlf             ;crlf を画面に出力 (09H)
          parse       reply[2], fcb[16]
                                       ; ファイル名の解析 (29H)
          rename      fcb              ; ファイル名の変更
```

INT 21H

ファンクション 19H カレントドライブ番号の取得

コール　AH = 19H

リターン　AL =現在選択されているドライブ（00H = A：、01H = B：、…）

解　説

ALに現在選択されているドライブ（00H = A：、01H = B：、…）が返されます。

マクロ定義

```
current_disk    macro
                mov     ah, 19H
                int     21H
                endm
```

サンプル

次のプログラムは、現在選択されている（カレント）ドライブを画面に表示します。

```
message     db  "Current disk is$"      ;"$"の説明は
crlf        db  13, 10, "$"             ; ファンクション 09H を参照
;
func_19H:   display  message            ;message を画面に出力
            current_disk                ; カレントドライブを得る
            add      al,'A'             ; 数値をキャラクタに変換
            display_char al             ; ドライブ名を画面に出力
            display_char ':'            ;
            display  crlf               ;crlf を画面に出力 (09H)
```

INT 21H

ファンクション 1AH　ディスク転送アドレスの設定

コール　AH　＝1AH
DS：DX＝ディスク転送アドレス

リターン　なし

解　　説

ディスクからの読み出し／書き込み（ファンクション 14H,15H,21H,22H,27H,28H）におけるディスク転送アドレス（ディスクバッファの位置）を指定します。また、ディレクトリの検索（ファンクション 11H,12H,4EH,4FH）を行う場合に返されるディレクトリ情報の格納先のアドレスを設定することができます。

DX には、ディスク転送アドレスのオフセット（DS にはセグメントアドレス）が入っていなければなりません。セグメントの終わりから先頭へ向うディスク転送や、他のセグメントをオーバーフローするようなディスク転送は許されていません。

注意　MS–DOS では、ディスク転送アドレスを設定しないと、PSP 内のオフセット 80H をデフォルト値として使用します。
カレントディスク転送アドレスは、ファンクション 2FH で得ることができます。

マクロ定義

```
set_dta     macro   buffer
            mov     dx, offset buffer
            mov     ah, 1AH
            int     21H
            endm
```

サンプル　次のプログラムはプロンプトを出力し、入力したアルファベットを数字（A＝01H、B＝02H、…）に変換して、次にドライブ B の〝ALPHABET.DAT〞という名前のファイルから対応するレコードを読み出し、それを画面に表示します。このファイルには 26 個のレコードが入り、1 個のレコードのサイズは 28 バイト長です。

```
record_size equ    14                ;FCB のレコードの
```

```
                                                ; サイズフィールドのオフセット
relative_record   equ      33                   ;FCB のレコードの
                                                ; 相対レコードフィールドのオフセット
;
fcb               db       2, "ALPHABETDAT"
                  db       25 dup(?)
buffer            db       34 dup(?), "$"
prompt            db       "Enter letter: $"
crlf              db       13, 10, "$"
;
func_1AH:    set_dta  buffer               ; ディスク転送アドレスの設定
             open     fcb                  ;ALPHABET.DAT のファイルの
                                           ; オープン (0FH)
             mov      fcb[record_size], 28 ; レコードサイズ 28 を設定
get_char:    display  prompt               ;prompt を画面に表示 (09H)
             read_kbd_and_echo             ; キーボード入力 (01H)
             cmp      al, 0DH              ; キャリッジリターンか?
             je       all_done             ; はいのとき、all_done へ
             sub      al, 41H              ; いいえのとき、ASCII コードを
                                           ; レコード番号に変換
             mov      fcb[relative_record], al
                                           ; 対応するレコードを設定
             display  crlf                 ;crlf を画面に出力
             read_ran fcb                  ;ALPHABET.DAT ファイルを
                                           ; ランダムリード (21H)
             display  buffer               ;buffer を画面に表示 (09H)
             display  crlf                 ;crlf を画面に出力 (09H)
             jmp      get_char             ; 次の文字を得る
all_done:    close    fcb                  ; ファイルをクローズ (15H)
```

INT 21H

ファンクション 1BH　カレントドライブのデータの取得

コール　AH＝1BH

リターン

AL　　＝1クラスタ当りのセクタ数
CX　　＝1セクタ当りのバイト数
DX　　＝1ドライブ当りのクラスタ数
DS：BX＝FAT–IDのアドレス

解　説

カレントドライブの各情報を以下のようにレジスタに返します。

AL=1クラスタ当りのセクタ数（アロケーションユニット）
CX=1セクタ当りのバイト数
DX=カレントドライブのクラスタ数

BXは、ドライブのタイプを表すファイルアロケーションテーブル（FAT）の、最初の1バイトのオフセットアドレス（DSは、セグメントアドレス）を返します。次にその内容を示します。

値	ドライブのタイプ
FFH	320Kバイトディスク、1トラック8セクタ
FEH	256Kバイトディスク、1トラック26セクタ
	1Mバイトディスク、1トラック8セクタ
	160Kバイトディスク、1トラック8セクタ
FDH	320Kバイトディスク、1トラック9セクタ
FCH	160Kバイトディスク、1トラック9セクタ
FBH	640Kバイトディスク、1トラック8セクタ
F9H	640Kバイトディスク、1トラック9セクタ
FEH	固定ディスク、または光ディスク

似た機能をもつファンクションが2つあります。1つはファンクション36H（ディスクのフリースペースを得る）で、違う点として、DXの返す値がFAT–IDのアドレスではなく、使用可能なクラスタ数になっています。もう1つは、ファンクション1CH（ドライブのデータを得る）で、カレントディスク以外のディスクのデータを得ることができます。

ファイルアロケーションテーブルを含むMS–DOSのディスクデータの詳細に関しては、第3章「MS–

DOS技術資料」を参照してください。

マクロ定義

```
def_drive_data  macro
                push    ds
                mov     ah, 1BH
                int     21H
                mov     al, byte ptr[bx]
                pop     ds
                endm
```

サンプル

次のプログラムは、デフォルトのドライブが1MバイトFDか別のディスクドライブかを判別します。

```
stdout      equ     1
;
msg         db      "Default drive is"
other       db      "another."
fd1M        db      "fd1M."
crlf        db      0DH, 0AH
;
func_1BH:  write_handle   stdout, msg, 17 ; メッセージ表示
           jc             write_error     ; エラー処理へ
           def_drive_data                ; デフォルトドライブのデータを得る
           cmp            byte ptr[bx], 0FEH ;FAT ID のリターン値=FEH か?
           jne            diskette        ; いいえのとき、diskette へ
           cmp            al, 8           ;1 クラスタあたり 8 セクタか
           jne            diskette        ; いいえのとき、diskette へ
           write_handle   stdout, fd1M, 5 ;fd1M を表示 (40H)
           jc             write_error     ; エラーのとき、write_error へ
           jmp short      all_done        ; クリア&all_done へ
diskette:  write_handle   stdout, other, 8 ;other を表示 (40H)
all_done:  write_handle   stdout, crlf, 2 ;crlf を表示 (40H)
           jc             write_error     ; エラー処理へ
```

INT 21H

ファンクション 1CH　ドライブのデータの取得

コール

AH ＝ 1CH
DL ＝ドライブ番号（00H ＝カレント、01H ＝ A：、02H ＝ B：…）

リターン

AL　＝ FFH　ドライブ番号の指定が無効
　　＝ FFH 以外　1クラスタ当りのセクタ数
CX　＝ 1セクタ当りのバイト数
DX　＝ 1ドライブ当りのクラスタ数
DS：BX ＝ FAT-ID のアドレス

解　説

DL で指定されたドライブ（00H ＝カレント、01H ＝ A：、…）の各情報を以下のようにレジスタに返します。

AL ＝ 1クラスタ当りのセクタ数（アロケーションユニット）当りのセクタ数
CX ＝ 1セクタ当りのバイト数
DX ＝ドライブのクラスタ数

BX は、ドライブのタイプを表すファイルアロケーションテーブル（FAT）の、最初の1バイトのオフセットアドレス（DS は、セグメントアドレス）を返します。次に、その内容を示します。

値	ドライブのタイプ
FFH	320K バイトディスク、1トラック 8 セクタ
FEH	256K バイトディスク、1トラック 26 セクタ
	1M バイトディスク、1トラック 8 セクタ
	160K バイトディスク、1トラック 8 セクタ
FDH	320K バイトディスク、1トラック 9 セクタ
FCH	160K バイトディスク、1トラック 9 セクタ
FBH	640K バイトディスク、1トラック 8 セクタ
F9H	640K バイトディスク、1トラック 9 セクタ
FEH	固定ディスク、または光ディスク

DL で指定されたドライブ番号が無効であると、AL に FFH を返します。
似た機能をもつファンクションが2つあります。1つは、ファンクション 36H（ディスクのフリース

ペースを得る）で、違う点は、BX の返す値が FAT–ID のアドレスではなく、使用可能なクラスタ数になっています。もう 1 つは、ファンクション 1BH（デフォルトドライブのデータを得る）で、デフォルトのディスクだけのデータを得ます。

ファイルアロケーションテーブルを含む MS–DOS のディスクデータの詳細に関しては、第 3 章「MS–DOS 技術資料」を参照してください。

マクロ定義

```
drive_data  macro   drive
            push    ds
            mov     dl, drive
            mov     ah, 1CH
            int     21H
            mov     al, byte ptr[bx]
            pop     ds
            endm
```

サンプル

次のプログラムは、ドライブ B が 1M バイトタイプか別のディスクドライブかを判別します。

```
stdout     equ     1
;
msg        db      "Drive B is"
other      db      "another."
fd1M       db      "fd1M."
crlf       db      0DH, 0AH
;
begin:     write handle stdout, msg, 11    ;msg を表示
           jc           write_error        ; エラー処理へ
           drive_data   2                  ; ドライブのデータを得る
           cmp          byte ptr[bx], 0FE  ;FAT ID のリターン値＝ 0FEH か?
           jnz          diskette           ; いいえのとき、diskette へ
           cmp          al, 8              ;1 クラスタあたり 8 セクタか
           jnz          diskette           ; いいえのとき、diskette へ
           jne          diskette
           write_handle stdout, fd1M, 5    ;fd1M を表示 (40H)
           jc           write_error        ; エラーの処理へ
           jmp          all_done           ; クリア&all_done へ
diskette:  write_handle stdout, other, 8   ;other を表示 (40H)
all_done:  write_handle stdout, crlf, 2    ;crlf を表示 (40H)
           jc           write_error        ; エラー処理へ
```

INT 21H

ファンクション 21H ランダムな読み出し

コール

AH ＝21H
DS：DX＝オープンされた FCB

リターン

AL ＝00H	読み出しは正常に行われ、処理が完了した
＝01H	ファイルの終わり（EOF）。または空レコード
＝02H	ディスク転送アドレス（DTA）に十分な空き領域がないため、読み込みは中止された
＝03H	ファイルの終わり（EOF）。レコードの残りの部分は、0で埋められた

解　説

相対レコードフィールドで指定したレコードを DTA に読み込みます。
DX には、オープンされた FCB のオフセット（DS にはセグメントアドレス）が入っていなければなりません。カレントブロック（オフセット 0CH）とカレントレコード（オフセット 20H）フィールドが、相対レコードフィールド（オフセット 21H）と一致するように設定され、これらのフィールドによって指定されたレコードが、ディスク転送アドレスにロードされます。

マクロ定義

```
read_ran    macro   fcb
            mov     dx, offset fcb
            mov     ah, 21H
            int     21H
            endm
```

サンプル

次のプログラムは文字入力を促すプロンプトを表示し、入力したアルファベットを数字（A = 01H、B = 02H、C = 03H、…）に変換して、ドライブBのALPHABET.DATという名前のファイルから対応するレコードを読み出し、それを画面に出力します。このファイルには26個のレコードが入り、1個のレコードのサイズは28バイト長です。

```
record_size      equ     14        ;FCBの
                                   ;サイズフィールドのオフセット
relative_record equ     33        ;FCBの
                                   ;相対レコードフィールドのオフセット
;
fcb              db      2, "ALPHABETDAT"
                 db      25 dup(?)
buffer           db      34 dup(?), "$"
prompt           db      "Enter letter: $"
crlf             db      13, 10, "$"
;
func_21H:   set_dta buffer          ;ディスク転送アドレスの設定（1AH)
            open fcb                ;ALPHABET.DATファイルのオープン
                                    ;（0FH)
            mov  fcb[record_size], 28  ;レコードサイズ28を設定
get_char:   display prompt          ;promptを画面に表示（09H)
            read_kbd_and_echo       ;キーボード入力（01H)
            cmp  al, 0DH            ;キャリッジリターンか?
            je   all_done           ;はいのとき、all_doneへ
            sub  al, 41H            ;いいえのとき、ASCIIコードを
                                    ;レコード番号に変換
            mov  fcb[relative_record], al  ;対応するレコードを設定
            display crlf            ;crlfを画面に出力（09H)
            read_ran fcb            ;ALPAHBET.DATファイルを
                                    ;ランダムな読み出し
            display buffer          ;bufferを画面に表示（09H)
            display crlf            ;crlfを画面に出力（09H)
            jmp     get_char        ;次の文字を得る
all_done:   close   fcb             ;ファイルをクローズ（10H)
```

INT 21H

ファンクション
22H ランダムな書き込み

コール

AH　　＝22H
DS：DX＝オープンされた FCB

リターン

AL ＝00H　書き込みは正常に行われ、処理が完了した
　＝01H　ディスクに空き領域がない
　＝02H　ディスク転送アドレス（DTA）に十分な空き領域がないため、書き込みは中止された

解　説

相対レコードフィールドで指定したレコードに DTA にあるデータを書き込みます。

DX には、オープンされた FCB 内のオフセット（DS にはセグメントアドレス）が入っていなければなりません。カレントブロック（オフセット 0CH）とカレントレコード（オフセット 20H）フィールドが相対レコードフィールド（オフセット 21H）と一致するように設定され、次にこれらのフィールドによって指定されたレコードへ、ディスク転送アドレスから書き込まれます。レコードサイズが 1 セクタよりも小さいと、ディスク転送アドレスにあるデータがバッファに移され、このバッファに入れられたデータが 1 セクタに達すると、ファイルのクローズか、ディスクのリセットシステムコール（ファンクション 0DH）の実行によって、このバッファがディスクに書き込まれます。

マクロ定義

```
write_ran   macro   fcb
            mov     dx, offset fcb
            mov     ah, 22H
            int     21H
            endm
```

サンプル

次のプログラムは文字入力を促すプロンプトを表示し、入力したアルファベットを数字（A = 01H、B = 02H、C = 03H、…）に変換して、次にドライブBのALPHABET.DATという名前のファイルから対応するレコードを読み込み、それを画面に出力します。このファイルには26個のレコードが入り、1個のレコードのサイズは28バイト長です。該当するレコードを出力すると、変更されたレコードを入力させるためにプロンプトを出力します。ユーザーが新規のレコードを入力すると、そのレコードはファイルに書き込まれます。リターンキーだけを押すと、レコードの置換は行われません。

```
record_size      equ      14        ;FCB の
                                    ; サイズフィールドのオフセット
relative_record equu     33        ;FCB の
                                    ; 相対レコードフィールドのオフセット
;
fcb              db     2, "ALPHABETDAT"
                 db     25 dup(?)
buffer           db     26 dup(?), 13H, 10H, "$"
prompt1          db     "Enter letter: $"
prompt2          db     "New record (RETURN for no change) : $"
crlf             db     13, 10, "$"
reply            db     28 dup(32)
blanks           db     26 dup(32)
;
func_22H:    set_dta buffer         ; ディスク転送アドレスの設定 (1AH)
             open    fcb            ;ALPHABET.DAT ファイルのオープン (0FH)
             mov     fcb[record_size], 28   ; レコードサイズ 28 を設定
get_char:    display prompt1        ;prompt1 を画面に表示 (09H)
             read_kbd_and_echo      ; キーボード入力 (01H)
             cmp     al, 0DH        ; キャリッジリターンか?
             je      all_done       ; はいのとき、all_done へ
             sub     al, 41H        ; いいえのとき、ASCII コードを
                                    ; レコード番号に変換
             mov     fcb[relative_record], al
                                    ; 対応するレコードを設定
             display crlf           ;crlf を画面に出力 (09H)
             read_ran fcb           ; ランダムな書き込み
             display buffer         ;buffer を画面に表示 (09H)
             display crlf           ;crlf を画面に出力 (09H)
             display prompt2        ;prompt2 を画面に表示 (09H)
             get_string 27, reply   ; バッファードキーボード入力 (0AH)
             display crlf           ;crlf を画面に出力 (09H)
```

```
            cmp replay[1], 0       ; キャリッジリターン以外のキーが
                                   ; 押されたか?
            je      get_char       ; いいえのとき、
                                   ; 次の文字を得る
            xor     bx, bx
            mov     bl, reply[1]       ; カウンタとして reply の
                                       ; バッファレングスを使用
            move_string blanks, buffer, 26        ; 章末参照
            move_string reply[2], buffer, bx      ; 章末参照
            write_ran   fcb          ; ランダムな書き込み
            jmp         get_char     ; 次の文字を得る
all_done:   close       fcb          ; ファイルをクローズ (10H)
```

22H

INT 21H

ファンクション **23H**

ファイルの大きさの取得

コール

AH　　=23H
DS：DX=オープンされていないFCB

リターン

AL =00H　ディレクトリエントリが存在する
　 =FFH　ディレクトリエントリが存在しない

解　　説

指定したディレクトリエントリのレコードサイズフィールドから、ファイルサイズを算出します。DXには、オープンされていないFCBのオフセット（DSには、セグメントアドレス）が入っていなければなりません。このファンクションコールを行うには、事前にレコードサイズフィールド（オフセット0EH）を該当する値に設定しておきます。最初に一致するエントリを見つけるために、このディスクディレクトリが検索されます。

一致するディレクトリエントリが存在すると、相対レコードフィールド（オフセット21H）が、ディレクトリ内のファイルサイズ（オフセット1CH）とFCB内のレコードサイズフィールド（オフセット0EH）から計算したファイルのレコード数に設定され、ALに00Hが返されます。

一致するディレクトリが存在しないと、ALにFFH（255）が返されます。

注意　FCBのレコードサイズフィールド（オフセット0EH）の値が、レコード内の実際の文字数と一致しない場合、このファンクションは正しいファイルサイズを返しません。デフォルトレコードサイズ（128）を使わないとき、このファンクションを使用する前にレコードサイズフィールドを正しい値に設定しておかなければなりません。

マクロ定義

```
file_size    macro    fcb
             mov      dx, offset fcb
             mov      ah, 23H
             int      21H
             endm
```

サンプル

次のプログラムは、ファイル名の入力を促すプロンプトを表示し、このファイルをオープンしてFCB内のレコードサイズフィールドを埋め、ファイルサイズシステムコールを行って、ファイルサイズとレコード数を16進で画面に表示します。

```
fcb          db       37 dup(?)
prompt       db       "File  name: $"
msg1         db       "Record length: ", 13, 10, "$"
msg2         db       "Records:       ", 13, 10, "$"
crlf         db       13, 10, "$"
reply        db       17  dup(?)
sixteen      db       16
;
func_23H:    display prompt              ;prompt を画面に表示 (09H)
             get_string 17, reply        ; バッファードキーボード入力 (0OH)
             cmp      reply[1], 0        ; キャリッジリターンか?
             jne      get_length         ; いいえのとき、get_length へ
             jmp      all_done           ; はいのとき、all_done へ
get_length: display crlf                ;crlf を画面に出力 (09H)
             parse    reply[2], fcb      ; ファイル名の解析 (29H)
             open     fcb                ; ファイルのオープン (0FH)
             file_size fcb               ; ファイルの大きさを得る
             mov      si, 33             ; 相対レコードフィールドの
                                         ; オフセットを設定
             mov      di, 9              ;msg2 に答える
convert_it: cmp      fcb[si], 0         ; 変換する数字か?
             je       show_it            ; いいえのとき、show_it へ
             convert fcb[si], sixteen, msg2[di]
             inc      si                 ;n-o-r インディックスをインクリメント
             inc      di                 ; メッセージインデックスを
                                         ; インクリメント
             jmp      convert_it         ; 数字をチェック
show_it:     convert fcb[14], sixteen, msg1[15]
             display msg1                ;msg1 を画面に表示 (09H)
             display msg2                ;msg2 を画面に表示 (09H)
             jmp      func_23H           ; 別のファイル名を得る
all_done:    close    fcb                ; ファイルをクローズ (10H)
```

INT 21H

ファンクション 24H 相対レコードの設定

コール　AH　　= 24H
DS：DX＝オープンされた FCB

リターン　なし

解　説

ランダムアクセスのときは、相対レコードフィールドの値によって、どのレコードに読み出し／書き込みを行うかを決定します。シーケンシャルアクセスのときは、カレントブロックとカレントレコードのふたつのフィールドの値から、次の式によって算出されるレコードに、読み出し／書き込みを行います。

(カレントブロック)*128＋カレントレコード

DX には、オープンされた FCB のオフセット（DS には、セグメントアドレス）が入っていなければなりません。相対レコードフィールド（オフセット 21H）は、カレントブロック（オフセット 0CH）、カレントレコードフィールド（オフセット 20H）と同じファイルアドレスに設定されます。

ファンクション 21H、22H、27H、28H を使う前に、このファンクションを使ってファイルポインタを設定しなければなりません。

マクロ定義

```
set_relative_record macro    fcb
                    mov      dx, offset  fcb
                    mov      ah, 24H
                    int      21H
                    endm
```

サンプル

次のプログラムは、ランダムなブロックの読み出し（読み出し）とランダムなブロックの書き込み（書き込み）システムコールを使用して、ファイルのコピーを行います。レコードサイズをファイルの大きさと等しくなるように設定し、レコードカウントを 1 に設定して 32K バイトのバッファを使用すると、コピー速度が速くなります。ファイルポインタは、カレントレコードフィールド（オフセット 20H）を 1 に設定し、相対レコードの設定機能を使い、相対レコードフィールド（オフセット 21H）をカレントブロック（オフセット 0CH）とカレントレコードフィー

ルド（オフセット 20H）を組み合わせたものと同じレコードにポイントさせることによって位置決めされます。

```
current_record  equ 32       ;FCB のレコードサイズフィールドの
                             ; オフセット
file_size       equ 16       ;FCB のレコードサイズフィールドの
                             ; オフセット
;
fcb             db     37  dup(?)
filename        db     17  dup(?)
prompt1         db     "File to copy: $" ;"$"の説明は
                                         ; ファンクション 09H を参照
prompt2         db     "Name of copy: $"
crlf            db     13, 10, "$"
file_length     dw     ?
buffer          db     32767 dup(?)
;
func_24H: set_dta buffer       ; ディスク転送アドレスの設定（1AH）
          display prompt 1     ;prompt1 を画面に表示（09H）
          get_string 15, filename      ; ファイルの名の入力（0AH）
          display crlf                 ;crlf を画面に出力（09H）
          parse filename[2], fcb       ; ファイル名の解析（29H）
          open  fcb                    ; ファイルのオープン（0FH）
          mov fcb[current_record], 0   ; 相対レコードフィールドを設定
          set_relative_record fcb          ; 相対レコードを設定
          mov ax, word ptr fcb[file_size] ; ファイルサイズを得る
          mov file_length, ax          ; ランダムなブロックの書き込みを
                                       ; するためにそれをセーブ
          ran_block_read fcb, 1, ax    ; ランダムなブロックの読み出し（27H）
          display prompt2              ;prompt2 を画面に表示（09H）
          get_string 15, filename      ; ファイルの名の入力（0AH）
          display crlf                 ;crlf を画面に出力（09H）
          parse filename[2], fcb       ; ファイル名の解析（29H）
          create fcb                   ; ファイルの作成（16H）
          mov fcb[current_record], 0   ; カレントレコードフィールド
                                       ; を設定
          set_relative_record fcb      ; 相対レコードを設定
          mov ax, file_length          ; オリジナルファイルの長さを得る
          ran_block_write fcb, 1, ax   ; ランダムなブロックの書き込み（28H）
          close fcb                    ; ファイルのクローズ（10H）
```

INT 21H

ファンクション 25H 割り込みベクタの設定

コール

AH　　＝25H
AL　　＝割り込みタイプ番号
DS：DX＝割り込み処理ルーチンのアドレス

リターン

なし

解　説

このファンクションは、任意の割り込みベクタを設定します。MS–DOS は、これにより、プロセスごとに割り込みを管理することができます。

指定した割り込みのベクタテーブルに、DS：DX で示される割り込み処理ルーチンのアドレスを設定します。DX には、割り込み処理ルーチンのオフセット（DX には、セグメントアドレス）が入っていなければなりません。AL には、このルーチンによって処理される割り込みタイプの番号が入っていなければなりません。

マクロ定義

```
set_vector  macro   interrupt, seg_addr, off_addr
            push    ds
            mov     ax, seg_addr
            mov     ds, ax
            mov     dx, off_addr
            mov     al, interrupt
            mov     ah, 25H
            int     21H
            pop     ds
            endm
```

サンプル

```
            lds   dx, intvector
            mov   ah, 25H
            mov   al, intnumber
            int   21H
      ; エラーがなければリターン
```

INT 21H

ファンクション 26H　新しいPSPの作成

コール　AH ＝ 26H
DX ＝新しい PSP のセグメントアドレス

リターン　なし

解　説

DX で指定したセグメントアドレスで、新しい PSP を作成します。

このファンクションコールは、バージョン 2.0 以前の MS-DOS と互換性を保つために用意されています。新しく作成するプログラムがバージョン 2.0 以前と互換性を保つ必要がなければ、ファンクション 4BH、コード 00H を使って子プロセスを起動してください。

マクロ定義

```
create_psp  macro   seg_addr
            mov     dx, seg_addr
            mov     ah, 26H
            endm
```

サンプル　このファンクションは、ファンクション 4BH、コード 00H（プログラムのロードと実行)、ファンクション 4BH、コード 03H（オーバーレイのロード）によって置き換えられるため、プログラムは省略します。

INT 21H

ファンクション 27H ランダムなブロックの読み出し

コール

AH　　＝27H
DS：DX＝オープンされた FCB
CX　　＝読み出すべきレコード数

リターン

AL ＝00H　読み出しは正常に行われ、処理が完了した
　＝01H　ファイルの終わり（EOF）。または空レコード
　＝02H　ディスク転送アドレス（DTA）に十分な空き領域がないため、読み出しは中止された。
　＝03H　ファイルの終わり（EOF）。レコードの残りの部分は、0で埋められた
CX ＝読み取られたレコード数

解　説

CX で指定したレコード数のデータを、ファイルから DTA に読み出します。

DX には、オープンされた FCB のオフセット（DS には、セグメントアドレス）が入っていなければなりません。CX には、読み出すべきレコード数を設定します。CX に 0 が入っていると、レコードは読み出されずに（動作が行われない）、このファンクションを終了します。指定されたレコード数（レコードサイズフィールド（オフセット 0EH）から計算される）の読み出しが、相対レコードフィールド（オフセット 21H）で指定されたレコードから開始されます。読み出されたレコードは、ディスク転送アドレスに入ります。

読み取られたレコード数が CX に返されます。カレントブロック（オフセット 0CH）、カレントレコード（オフセット 20H）、および相対レコード（オフセット 21H）フィールドは、次のレコードのアドレスに設定されます。

このファンクションの実行前に、ファンクション 24H によって相対レコードを設定しなければなりません。

マクロ定義

```
ran_block_read      macro fcb, count, rec_size
            mov     dx, offset fcb
            mov     cx, count
            mov     word ptr fcb[14], rec_size
            mov     ah, 27H
```

```
        int     21H
        endm
```

サンプル

次のプログラムは、ランダムなブロックの読み出しとランダムなブロックの書き込み（28H）のファンクションを使ってファイルをコピーします。レコードカウントをファイルの大きさと等しくなるように指定し、レコードサイズを1に指定して、32Kバイトのバッファを使用すると、コピーの速度を速くすることができます。ファイルの読み出しと書き込みは、1回のディスクアクセスで行われるので、コピーが高速になります（ファンクション27Hのプログラム例と比較してください。ファンクション27Hでは、レコードのカウントが1に、またレコードサイズがファイルの大きさと等しくなるように指定されています）。


```
current_record  equ     32          ;カレントレコードフィールドのオフセット
file_size       equ     16          ;ファイルサイズフィールドのオフセット
;
fcb             db      37 dup(?)
filename        db      17 dup(?)
prompt1         db      "File to copy: $"  ;"$"の説明はファンクション
prompt2         db      "Name of copy: $"  ;09Hを参照
crlf            db      13, 10, "$"
num_recs        dw      ?
buffer          db      32767 dup(?)
;
func_27H:       set_dta buffer              ;ディスク転送アドレスの設定（1AH）
                display prompt1             ;prompt1を画面に表示（09H）
                get_string 15, filename     ;ファイル名の入力（0AH）
                display crlf                ;crlfを画面に出力（09H）
                parse   filename[2], fcb    ;ファイル名の解析（29H）
                open    fcb                 ;ファイルのオープン（0FH）
                mov     fcb[current_record], 0
                                            ;カレントレコードフィールドに
                                            ;0を設定
                set_relative_record fcb     ;相対レコードを設定（24H）
                mov     ax, word ptr fcb[file_size]
                                            ;ファイルサイズを得る
                mov     num_recs, ax        ;ランダムなブロックの書き込み
                                            ;のためにそれをセーブ
                ran_block_read  fcb, num_recs, 1
                                            ;ランダムなブロックの読み出し
                display prompt2             ;prompt2を画面に表示（09H）
                get_string 15, filename     ;ファイル名の入力（0AH）
                display crlf                ;crlfを画面に出力（09H）
```

```
parse    filename[2], fcb ; ファイル名の解析（29H）
create   fcb              ; ファイルの作成（16H）
mov      fcb[current_record], 0
                          ; カレントレコードフィールドに
                          ;0 を設定
set_relative_record  fcb ; 相対レコードを設定（24H）
mov      ax, file_length  ; オリジナルファイルのサイズを得る
ran_block_write fcb, num_recs, 1
                          ; ランダムなブロックの書き込み
close    fcb              ; ファイルをクローズ（10H）
```

INT 21H

ファンクション 28H　ランダムなブロックの書き込み

コール

AH　　=28H
DS：DX=オープンされたFCB
CX　　=書き込むべきレコード数（0=ファイルサイズフィールドを設定します。）

リターン

AL =00H　書き込みは正常に行われ、処理が完了した
　=01H　ディスクの空き領域がない
　=02H　ディスク転送アドレス（DTA）に十分な空き領域がないため、書き込みは中止された

CX =書き込まれたレコード数

解　説

CXで指定したレコード数のデータが、DTAからファイルに書き込まれます。

DXにはオープンされたFCBのオフセット（DSには、セグメントアドレス）が、CXには書き込むべきレコード数、または0が入っていなければなりません。指定されたレコード数（オフセット0EHのレコードサイズフィールドから計算する）が、ディスク転送アドレスから書き込まれます。ファイルへのレコードの書き込みは、FCBの相対レコードフィールド（オフセット21H）で指定されたレコードから開始されます。CXが0であると、レコードは書き込まれませんが、ディレクトリエントリのファイルサイズフィールド（オフセット1CH）が、FCBの相対レコードフィールド（オフセット21H）で指定されたレコード数に設定されます。アロケーションユニットは、必要に応じ割り当てられるか、または開放されます。

このファンクションの実行前に、ファンクション24Hによって相対レコードを設定しなければなりません。

書き込まれたレコード数がCXに返されます。カレントブロック（オフセット0CH）、カレントレコード（オフセット20H）および相対レコード（オフセット21H）の各フィールドは、その次のレコードアドレスに設定されます。

マクロ定義

```
ran_block_write macro   fcb, count, rec_size
                mov     dx, offset fcb
                mov     cx, count
```

```
        mov     word ptr fcb[14], rec_size
        mov     ah, 28H
        int     21H
        endm
```

サンプル　ファンクション27Hを参照してください。

INT 21H

ファンクション
29H　ファイル名の解析

コール

AH　= 29H
AL　=解析の制御（解説を参照）
DS：SI =解析すべき文字列
ES：DI =オープンされていない FCB

リターン

AL　= 00H　ワイルドカード文字が、使用されていない
　　= 01H　ワイルドカード文字が、使用されている
　　= FFH　ドライブ文字が無効
DS：SI =解析された文字列の次にくる最初のバイト
ES：DI =オープンされていない FCB

解　説

DS：SI で指定したアドレスからはじまる〝d：ファイル名.拡張子〟という書式のファイル名の文字列を、ES：DI で指定したアドレスにオープンされていない FCB の形式でセットします。

SI には解析すべき文字列（コマンド行）のオフセット、DS にはセグメントアドレスが、DI にはオープンされていない FCB のオフセット（ES には、セグメントアドレス）が入っていなければなりません。

AL レジスタの 0〜3 ビット目は、解析処理を制御するためのものです。4〜7 ビット目は、無視されます。

ビット	値	意　味
0	0	ファイル分離記号を検出した場合、すべての解析を停止。
	1	先行する分離記号を無視。
1	0	文字列にドライブ番号が入っていない場合、FCB 内のドライブ番号は 0（カレントドライブ）に設定される。
	1	文字列にドライブ番号が入っていない場合、FCB 内のドライブ番号は変更されない。
2	1	文字列にファイル名が入っていない場合、FCB 内のファイル名は変更されない。
	0	文字列にファイル名が入っていない場合、FCB 内のファイル名は 8 つのスペースに設定される。

ビット	値	意　味
3	1	文字列に拡張子が入っていない場合、FCB内の拡張子は変更されない。
	0	文字列に拡張子が入っていない場合、FCB内の拡張子は3つのスペースに設定される。

ファイル名か拡張子にアスタリスク（*）が入っていると、ファイル名または拡張子内の、他のすべての文字は疑問符（?）に設定されます。

次に、ファイル名分離記号を示します。

：　.　;　,　=　+　/　”　[　]　¥　<　>　|　スペース　タブ

ファイル名の終了記号には、すべてのファイル名の分離記号と、すべての制御文字が含まれます。ファイル名の中にファイル名の終了記号を入れることはできません。終了記号を検出すると、解析が停止します。

文字列に有効なファイル名が入っている場合

1……　ファイル名または拡張子にワイルドカード文字（*または?）が入っていると、ALに1が、入っていないときは0がALに返される。

2……　DS：SIは、解析された文字列の、次の最初の文字を示す。ES：DIは、オープンされていないFCBの先頭のバイトを示す。

ドライブ名が無効であると、ALにFFHが返されます。文字列に有効なファイル名が入っていないと、ES：DI+1はスペース（ASCIIコード32）を示します。

マクロ定義

```
parse   macro   string, fcb
        mov     si, offset string
        mov     di, offset fcb
        push    es
        push    ds
        pop     es
        mov     al, 0FH        ;0、1、2、3のビットがONである
        mov     ah, 29H
        int     21H
        pop     es
        endm
```

サンプル

次のプログラムは、プロンプトで入力された名前のファイルが、存在するかどうかを検索します。

```
fcb         db      37 dup(?)
prompt      db      "Filename: $"
reply       db      17 dup(?)
yes         db      "FILE EXISTS", 13, 10, "$"
no          db      "FILE DOES NOT EXIST", 13, 10, "$"
;
func_29H:   display prompt                ;prompt を画面に表示 (09H)
            get_string    15, reply      ; ファイル名の入力 (0AH)
            parse         reply[2], fcb  ; ファイル名の解析
            search_first  fcb            ; 最初のエントリを検索 (11H)
            cmp           al, 0FFH       ; ディレクトリエントリが存在するか?
            je            not_there      ; いいえのとき、not_there へ
            display       yes            ; はいのとき、yes を画面に表示 (09H)
            jmp           continue
not_there:  display       no
continue:   .
            .
```

INT 21H

ファンクション 2AH 日付の取得

コール　AH ＝ 2AH

リターン　CX ＝年（1980〜2079）
DH ＝月（1〜12）
DL ＝日（1〜31）
AL ＝曜日（0 ＝日、1 ＝月、…、6 ＝土）

解　　説

CX と DX に現在の日付が 2 進数でシステムから返され、AL には曜日が返されます。

マクロ定義

```
get_date    macro
            mov  ah, 2AH
            int    21H
            endm
```

サンプル

次のプログラムは日付を取得し、翌日の日付に更新します。必要に応じて、月または年を 1 つ増やし、新しい日付に設定します。

```
month       db  31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
;
func_2AH:   get_date              ; 日付を得る
            inc  dl               ; 日をインクリメント
            xor  bx, bx           ;BL はインデックスとして使用
            mov  bl, dh           ; 月をインデックスレジスタに設定
            dec  bx
            cmp  dl, month[bx]    ; 月の最後の日を越えているか?
            jle  month_ok         ; いいえのとき、新規の日付を設定、
                                  ;month_ok へ
            mov  dl, 1            ; はいのとき、日を 1 に設定
            inc  dh               ; そして、月をインクリメント
            cmp  dh, 12           ; 年の最後の月を越えているか?
```

```
            jle   month_ok          ; いいえのとき、新規の日付を設定、
                                    ;month_ok へ
            mov   dh, 1             ; はいのとき、月を 1 に設定
            inc   cx                ; 年をインクリメント
month_ok:   set_date cx, dh, dl     ; 日付の設定 (2BH)
```

INT 21H

ファンクション 2BH　日付の設定

コール

AH ＝2BH
CX ＝年（1980〜2079）
DH ＝月（1＝1月、2＝2月、…）
DL ＝日（1〜31）

リターン

AL ＝00H　有効な日付
　 ＝FFH　無効な日付

解　説

CX と DX に 2 進数で指定した年月日を、システムのカレンダーに設定します。

日付が有効であると、この日付が設定され AL に 00H が返されます。無効であると、このファンクションは中止され、AL に FFH（255）が返されます。

マクロ定義

```
set_date    macro   year, month, day
            mov     cx, year
            mov     dh, month
            mov     dl, day
            mov     ah, 2BH
            int     21H
            endm
```

サンプル

次のプログラムは日付を取得し、翌日の日付に更新します。必要に応じて、月または年を1つ増やし、新しい日付に設定します。

```
month       db    31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
;
func_2BH:   get_date                ; 日付を得る（2AH）
            inc     dl              ; 日をインクリメント
            xor     bx, bx          ;BL はインデックスとして使用
            mov     bl, dh          ; 月をインデックスレジスタに設定
            dec     bx
```

```
              cmp      dl, month[bx] ; 月の最後の日を越えているか?
              jle      month_ok      ; いいえのとき、新規の日付を設定、
                                     ;month_ok へ
              mov      dl, 1         ; はいのとき、日を 1 に設定
              inc      dh            ; そして、月をインクリメント
              cmp      dh, 12        ; 年の最後の月を越えているか?
              jle      month_ok      ; いいえのとき、新規の日付を設定、
                                     ;month_ok へ
              mov      dh, 1         ; はいのとき、月を 1 に設定
              inc      cx            ; 年をインクリメント
month_ok:     set_date cx, dh, dl    ; 日付の設定
```

INT 21H

ファンクション 2CH　時刻の取得

コール　AH ＝ 2CH

リターン　CH ＝時（0〜23）
CL ＝分（0〜59）
DH ＝秒（0〜59）

解　説

現在の時刻をシステム時計から2進数でCXとDXに返します。

マクロ定義

```
get_time    macro
            mov      ah, 2CH
            int      21H
            endm
```

サンプル　次のプログラムは、任意のキーが入力されるまで、継続的に時刻を画面に出力します。

```
time        db       "00:00:00.00", 13, 10, "$"
ten         db       10
.
func_2CH:   get_time                      ; 時刻を得る（このファンクション）
            convert ch, ten, time         ; 章末参照
            convert cl, ten, time[3]      ; 章末参照
            convert dh, ten, time[6]      ; 章末参照
            display time                  ; 時刻を画面に表示（09H）
            check_kbd_status              ; キーボードステータスの検査（0BH）
            cmp      al, 0FFH             ; キー入力されたか?
            je       all_done             ; はいのとき、処理終了
            jmp      func_2CH             ; いいえのとき、時刻の表示を継続
all_done:   .
            .
```

INT 21H

ファンクション 2DH 時刻の設定

コール

AH ＝ 2DH
CH ＝時（0～23）
CL ＝分（0～59）
DH ＝秒（0～59）
DL ＝ 00H

リターン

AL ＝ 00H　有効な時刻
　＝ FFH　無効な時刻

解説

CX と DX に 2 進数に設定した時刻をシステム時計にセットします。DL ＝ 00H でなければなりません。

指定された時刻が有効であると、その時刻が設定され AL に 00H が返されます。無効であると、このファンクションは中止され AL に FFH（255）が返されます。

マクロ定義

```
set_time    macro   hour, minutes, seconds
            mov     ch, hour
            mov     cl, minutes
            mov     dh, seconds
            mov     di, 0
            mov     ah, 2DH
            int     21H
            endm
```

サンプル

次のプログラムは、システムクロックを 0 に設定したのち、時刻を継続的に画面に出力します。任意のキーが入力されると時刻の表示が停止し、再びキーが入力されると、クロックは 0 にリセットされ時刻の表示が再開します。

```
time        db      "00:00:00.00", 13, 10, "$"
ten         db      10
;
```

```
func_2DH:    set_time  0, 0, 0              ; 時刻を設定
read_clock:  get_time                       ; 時刻を得る (2CH)
             convert ch, ten, time          ; 章末参照
             convert cl, ten, time[3]       ; 章末参照
             convert dh, ten, time[6]       ; 章末参照
             display time                   ; 時刻を画面に表示 (09H)
             dir_console_io 0FFH            ; キー入力 (06H)
             cmp      al, 00H               ; 文字は入力されたか?
             jne      stop                  ; はいのとき、時刻の表示を
                                            ; 停止、stop へ
             jmp      read_clock            ; いいえのとき、時刻の表示を
                                            ; 継続
stop:        read_kbd                       ; キーの再入力 (08H)
             jmp      func_2DH              ; 時刻の表示を再開
```

INT 21H

ファンクション
2EH ベリファイフラグのセット/リセット

コール

AH = 2EH
AL = 00H ベリファイを行わない
= 01H ベリファイを行う
DL = 00H

リターン

なし

解説

ALには、01H（ディスクへ書き込むたびに、ベリファイを行う）または00H（ベリファイなしで書き込みを行う）のいずれかを、DLには00Hをセットします。MS–DOSでは、ディスクに書き込みが行われるたびに、このフラグの検証を行います。

重要なデータをディスクに書き込む場合、このフラグをオンにした方がよいでしょう。ただし、ディスクエラーが発生するのはまれであり、ベリファイを行うと処理速度が遅くなるため、通常のデータを処理するときは、オフにしてもよいでしょう。

マクロ定義

```
verify      macro   switch
            mov     al, switch
            mov     ah, 2EH
            mov     dl, 0
            int     21H
            endm
```

INT 21H

ファンクション
2FH ディスク転送アドレスの取得

コール　AH＝2FH

リターン　ES：BX＝ディスク転送アドレス

解　説

ディスク転送アドレスのセグメントをESに、オフセットをBXに返します。エラーコードは返しません。

マクロ定義

```
get_dta     macro
            mov     ah, 2FH
            int     21H
            endm
```

サンプル　次のプログラムは、カレントディスクの転送アドレスを表示します。

```
message   db      "DTA--      :       ", 0DH, 0AH, "$"
sixteen   db      10H
temp      db      2 dup(?)
;
func_2FH: get_dta                          ;THIS FUNCTION
          mov     word ptr temp, ES        ;ディスク転送アドレスを得る
          convert temp[1], sixteen, message[07H]
                                           ;CONVERT については章末参照
          convert temp, sixteen, message[09H]
          convert bh, sixteen, message[0CH]
          convert bl, sixteen, message[0EH]
          display message                  ;message を画面に表示 (09H)
```

INT 21H

ファンクション
30H MS–DOS バージョン番号の取得

コール AH ＝30H

リターン
AL ＝バージョン番号の整数部
AH ＝バージョン番号の小数部
BH ＝FFH
BL：CX＝000000H

解説

MS–DOS のバージョン番号を返します。このとき、AL、AH には、それぞれのバージョン番号の整数部、小数部が入ります。たとえば、MS–DOS バージョン 3.3 の場合、AL は 3（03H）に、AH は 30（1EH）になります。AL が 0 の場合、MS–DOS バージョン 2.0 以前のバージョンを表します。

マクロ定義

```
get_version   macro, code
              mov     ah, 30H
              int     21H
              endm
```

INT 21H

ファンクション 31H　プロセスの常駐終了

コール

AH ＝ 31H
AL ＝抜け出しコード
DX ＝パラグラフ単位（16バイト単位）でのメモリサイズ

リターン

なし

解　説

メモリ上にプログラムを残したまま、プロセスを終了させます。また、デバイスの特殊な割り込みハンドルにも使用されることがあります。割り込みタイプ27Hと異なり、64Kバイト以上のプロセスの常駐を許し、CS（PSPのセグメントアドレス）の設定を必要としません。MS-DOSバージョン2.0との互換性を保つ必要があるような特別な場合を除いて、割り込みタイプ27Hではなく、このファンクション31Hを使用してください。

DXは、常駐するプログラムが必要とするメモリのパラグラフ数（1パラグラフ＝16バイト）でなければならず、EXE形式のプログラムの場合は特に注意が必要です。DXの値は、常駐するプログラムに100Hバイトのプログラムヘッダプレフィクスを加えたサイズでなければなりません。

MS-DOSは現在のプロセスを終了し、イニシャルアロケーションブロックをパラグラフの大きさでセットします。このコールは、このプロセスに属する他のアロケーションブロックを開放するものではありません。AL内に渡された抜け出しコードは、ファンクション4DHを通して、親プロセスから取得することができます。

マクロ定義

```
keep_process    macro   return_code, last_byte
                mov       al, return_code
                mov       dx, offset last_byte
                mov       cl, 4
                shr       dx, cl
                inc       dx
                mov       ah, 31H
                int       21H
                endm
```

サンプル

このコールの使い方のほとんどはマシンに依存するため、プログラムは省略します。マクロ定義を参考にしてください。

INT 21H

ファンクション 33H <CTRL–C>チェックのセット／リセット

コール

AH ＝33H
AL ＝00H　現在のステータスを得る
　＝01H　ステータスのセット
DL（セットする場合：AL　＝01H）
　＝00H　オフ
　＝01H　オン

リターン

DL ＝00H　オフ
　＝01H　オン
AL ＝FFH　エラー（コールしたときのALが00Hまたは01Hでない）

解説

MS–DOSの<CTRL–C>チェックのステータスを得るか、またはセットします。ALの値は次のいずれかでなければなりません。

AL＝0　DLに現在の<CTRL–C>チェックのステータスを返す。
AL＝1　DLの値で、<CTRL–C>チェックのステータスを設定する。

ALが0であると、DLは現在の<CTRL–C>チェックのステータスを返します。ALが1であると、DLの値はセットされる<CTRL–C>チェックのステータスです（DL＝0：オフ、DL＝1：オン）。ALが0または1でないとALはFFHを返し、<CTRL–C>チェックのステータスは影響を受けません。

MS–DOSは通常、01Hから0CHまでのファンクションコール動作を実行しているときだけ、<CTRL–C>のチェックを行いますが、<CTRL–C>のチェックがオンのとき、すべてのシステムコールに対してこのチェックを行わせることができます。たとえば、<CTRL–C>のチェックがオフであると、すべてのディスクアクセスは、割り込みの実行に関係なく続けられますが、オンであると、ディスクアクセスを開始させたシステムコールに対しても<CTRL–C>の割り込みが実行されます。

注意　ファンクションコール06H、07Hによって、データとして<CTRL–C>を読み取るプログラムは、<CTRL–C>チェックがオフであることを確認する必要があります。

マクロ定義

```
ctrl_c_ck   macro    action, state
            mov      al, action
            mov      dl, state
            mov      ah, 33H
            int      21H
            endm
```

サンプル

次のプログラムは、<CTRL-C>チェックがオンかオフかのメッセージを表示します。

```
message     db          "Control-C checking", "$"
on          db          "on", "$", 0DH, 0AH, "$"
off         db          "off", "$", 0DH, 0AH, "$"
;
func_33H:   dispay      message    ;mesagge を表示 (09H)
            ctrl_c_ck   0          ; <CTRL-C >チェック
            cmp         dl, 0      ; オフか?
            jg          ck_on      ; いいえのとき、ck_on へ
            display     off        ; はいのとき、"off"を画面に表示 (09H)
            jmp         return     ; 処理終了
ck-on:      display     on         ;"on"を画面に表示 (09H)
```

INT 21H

ファンクション 35H 割り込みベクタの取得

コール

AH ＝35H
AL ＝割り込み番号

リターン

EX：BX＝割り込みルーチンのアドレス

解　　説

指定した割り込みの、割り込みベクタのアドレスを得ます。ALで、割り込み番号を指定します。BXには、割り込みハンドルのオフセットアドレス（ESはセグメントアドレス）が返されます。

互換性を保つため、割り込みベクタをメモリに直接読み書きしないでください。MS-DOSバージョン2.0との互換性を保つ必要があるような特別な場合を除いて、割り込みベクタを得るにはファンクション35Hを、割り込みベクタのセットにはファンクション25H（割り込みベクタのセット）を使用してください。

マクロ定義

```
get_vector  macro   interrupt
            mov     al, interrupt
            mov     ah, 35H
            int     21H
            endm
```

サンプル

次のプログラムは、割り込みタイプ25H（アブソリュートディスクリード）のアドレス（CS：IP）を表示します。

```
message     db          "Interrupt 25H-- CS：0000 IP:0000"
            db          0DH, 0AH, "$"
vec_seg     db          2 dup(?)
vec_off     db          2 dup(?)
;
func_35H: push        es                        ;ES をセーブ
          get_vector 25H                        ; 割り込みベクタを得る
          mov         ax, es                    ;INT25H のセグメントアドレス
                                                ; を AX にセット
```

```
        pop         es                  ;ESをリストア
        convert     ax, 16, message[20] ;章末参照
        convert     bx, 16, message[28] ;章末参照
        display     message             ;messageを画面に表示（09H）
```

INT 21H

ファンクション 36H ディスクのフリースペースの取得

コール

AH ＝ 36H
DL ＝ドライブ番号（00H ＝カレント、01H ＝ A：、02H ＝ B：、…）

リターン

BX ＝使用可能なクラスタ数
DX ＝ 1 ドライブ当たりの全クラスタ数
CX ＝ 1 セクタ当たりのバイト数
AX ＝ 1 クラスタ当たりのセクタ数
＝ FFFFH　ドライブ番号が無効

解　説

指定したドライブの使用可能なクラスタ数、ディスクのメディアの情報（計算によって使用可能なバイト数が得られます）を返します。DL で、ドライブを指定してください。ドライブ番号（00H ＝カレント、01H ＝ A：、…）が無効であると、AX は FFFFH を返します。

ファンクション 1BH、1CH は、バージョン 2.0 以前の MS-DOS と互換性を保つために用意されています。ファンクション 1BH、1CH の代わりに、このコールを使用してください。

マクロ定義

```
get_disk_space  macro   drive
                mov     dl, drive
                mov     ah, 36H
                int     21H
                endm
```

サンプル

次のプログラムは、ドライブ B のディスクのフリースペース情報を表示します。

```
message  db   "clusters on drive B.", 0DH, 0AH       ;DX
         db   "clusters available,", 0DH, 0AH       ;BX
         db   "sectors per cluster.", 0DH, 0AH       ;AX
         db   "bytes per sector,", 0DH, 0AH, "$"     ;CX
;
func_36H: get_disk_space  2                 ; ディスクのフリースペースを得る
          convert      ax, 10, message[55]  ; 章末参照
```

```
convert     bx, 10, message[28] ; 章末参照
convert     cx, 10, message[83] ; 章末参照
convert     dx, 10, message     ; 章末参照
display     message             ;message を画面に表示（09H）
```

INT 21H

ファンクション
38H 国別情報の取得

コール

AH ＝38H
AL ＝00H 現在の国
＝01H USA 規格
＝51H 日本規格
DS：DX＝32 バイトのメモリ領域に対するポインタ

リターン

キャリーフラグがセットされた場合
AX ＝02H 無効なファンクション（指定された国が見つからない）

キャリーフラグがセットされない場合
DS：DX に、国のデータがセットされる

解 説

このファンクション 38H は、MS–DOS がキーボード、画面の制御に使う国別情報を取得します。DX は 32 バイトの国別情報のメモリ領域のオフセットアドレス（セグメントアドレスは、DS で指定）でなければなりません。AL はカントリーコードで、次に、その内容を示します。

AL の値	意 味
0	現在の国の情報を取得する。
1〜FEH	このコードで指定された国の情報を取得する。

DS：DX でアドレスを指定された 32 バイトのメモリ領域の内容を次に示します。

オフセットアドレス	内 容
00H〜01H（2 バイト）	日付表示フォーマット
02H〜06H（5 バイト）	ASCIIZ 文字列・通貨記号
07H〜08H（2 バイト）	ASCIIZ 文字列・3 桁ごとの区切り記号
09H〜0AH（2 バイト）	ASCIIZ 文字列・10 進分離記号
0BH〜0CH（2 バイト）	ASCIIZ 文字列・日付分離記号
0DH〜0EH（2 バイト）	ASCIIZ 文字列・時刻分離記号
0FH（1 バイト）	ビットフィールド

オフセットアドレス	内　容
10H（1バイト）	通貨桁
11H(1バイト)	時刻フォーマット
12H～15H（4バイト）	ケースマッピングコール
16H～17H（2バイト）	ASCIIZ 文字列・データリスト分離記号

これらのエントリの大部分のフォーマットは、ASCIIZ（NUL コードで終了する ASCII 文字列）ですが、テーブルの索引を簡単にするため、割り当てられる各フィールドの大きさは固定されています。

日付の項目には、次のフォーマットで値が入ります。

0	USA 規格	m/d/y
1	ヨーロッパ規格	d/m/y
2	日本規格	y/m/d

ビットフィールドには、8ビットの値が入ります。現在定義されていないすべてのビットは、ランダムな値をもっていると想定しなければなりません。

0ビット目	＝0	通貨記号が金額の前に付く場合
	＝1	通貨記号が金額の後に付く場合
1ビット目	＝0	通貨記号が金額の直前に付く場合
	＝1	通貨記号と金額の間に、スペースを入れる場合

時刻フォーマットは、次の値が入ります。

0	12時間
1	24時間

通貨桁フィールドは、通貨金額の小数点以下の桁数を示します。

ケースマッピングコールとは、FAR 手続きのことで、これによって 80H から FFH までの文字に対し、国に固有の小文字から大文字のマッピングが行われます。このコールは、AL に入っているマップすべき文字を使用します。AL 内に文字が入っていると、この文字の正しい大文字コードが返されます。変更されるレジスタは、AL および FLAGS のみです。このルーチンに 80H 未満のコードを渡すことは可能ですが、この範囲の文字に対しては、何も行われません。この場合、マッピングは行われず、AL は変更されません。

マクロ定義

```
get_country     macro   country, buffer
                local   gc_01
                mov     dx, offset buffer
                mov     ax, country
gc_01H:         mov     ah, 38H
                int     21H
                endm
```

サンプル

次のプログラムは、時刻と日付を現在のカントリーコードで表示し、通貨記号と区切り記号を使って、999,999 と 99/100 を表示します。

```
time        db      "::", 5 dup(20H), "$"
date        db      " / / ", 5 dup(20H), "$"
number      db      "999?999?99", 0DH, 0AH, "$"
data_area   db      32 dup(?)
func_38H:   get_country 0, data_area        ; 国別情報を得る
            get_time                        ; 時刻を得る (2CH)
            byte_to_dec ch, time            ; 変換に関するマクロの説明は
            byte_to_dec cl, time[03H]       ; 章末を参照
            byte_to_dec dh, time[06H]
            get_date                        ; 日付を得る (2AH)
            sub         cx, 1900            ; 下 2 桁を得る
            byte_to_dec cl, date[06H]       ; 章末参照
            cmp         word ptr data_area, 0
                                            ; カントリーコードをチェック
            jne         not_usa             ;USA でないとき、not_usa へ
            byte_to_dec dh, date            ; 章末参照
            byte_to_dec dl, date[03H]       ; 章末参照
            jmp         all_done
not_usa:    byte_to_dec dl, date            ; 章末参照
            byte_to_dec dh, date[03H]       ; 章末参照
all_done:   mov         al, data_area[07H]  ;number に 3 桁ごとの区切
            mov         number[03H], al     ; り記号を入れる
            mov         al, data_area[09H]  ;AMOUNT に 10 進分離記号を
            mov         number[07H], al     ; 入れる
            display     time                ;time を画面に表示 (09H)
            display     date                ;date を画面に表示 (09H)
            display_char data_area[02H]     ; 文字を画面に表示 (02H)
            display     number              ;number を画面に表示 (09H)
```

INT 21H

ファンクション 38H　国別情報の設定

コール

AH ＝38H
DX ＝FFFFH
AL ＝FFH以外　カントリーコード
　＝FFH　BXにカントリーコードが入っている
BX（AL　＝FFHの場合）
　＝FFH以上のカントリーコード

リターン

キャリーフラグがセットされた場合
AX ＝02H　無効なカントリーコード（指定された国が見つからない）

キャリーフラグがセットされない場合
エラーなし

解　説

このファンクション38Hは、MS–DOSがキーボード、画面などの制御に使う国別情報をセットしたり、国別情報を取得します。DXは、FFFFH、つまり–1でなければなりません。ALはカントリーコードで、次にその内容を示します。

ALの値	意　味
01H〜FEH	このコードで指定された国のカントリーコード
FFH	BXで指定された国のカントリーコード

カントリーコードは、通常その国の国際電話プレフィクスコードです。
PC–9800シリーズではAL＝01H（USA規格）、AL＝51H（日本規格）のみ指定できます。
エラーがおこるとキャリーフラグがセットされ、AXにエラーコードを返します。

マクロ定義

```
set_country     macro   country
                local   sc_01
                mov     dx, 0FFFFH
                mov     ax, country
                cmp     ax, 0FFH
                jl      sc_01
```

```
                mov     bx, country
sc_01:          mov     ah, 38H
                int     21H
                endm
```

サンプル

次のプログラムは、カントリーコードをイギリス（44）に変えます。

```
            uk          equ         44
            ;
func_38H:   set_country uk  ;国別情報のカントリーコードをイギリスにセット
            jc          error
```

INT 21H

ファンクション 39H　ディレクトリの作成

コール

AH　　=39H
DS：DX=パス名の位置

リターン

キャリーフラグがセットされた場合
AX =03H　無効なパス
　 =05H　アクセスの否定（親ディレクトリ内に空きスペースがないか、すでに同名のディレクトリ／ファイルが存在しているため、ディレクトリが作成できなかった）

キャリーフラグがセットされない場合
エラーなし

解　説

新しいサブディレクトリを作成します。DX は、新しいサブディレクトリのパス名を表す ASCIIZ 文字列のオフセットアドレス（DS は、セグメントアドレス）でなければなりません。

エラーが起こるとキャリーフラグがセットされ、AX にエラーコードが返されます。

マクロ定義

```
make_dir    macro   path
            mov     dx, offset path
            mov     ah, 39H
            int     21H
            endm
```

サンプル

次のプログラムは、ドライブ B のディスク上のルートディレクトリに〝NEWDIR〟という名前のサブディレクトリを作成し、カレントディレクトリを一度〝NEWDIR〟に移してからルートディレクトリに戻り、〝NEWDIR〟を削除します。また、ディレクトリを移動するたびに、カレントディレクトリを表示します。

```
old_path    db      "b:¥", 0, 63 dup(?)
new_path    db      "b:¥newdir", 0
buffer      db      "b:¥", 0, 63 dup(?)
```

```
;
func_39H: get_dir    2, old_path[03H] ；カレントディレクトリ情報を得る
          jc                error_get
          display_asciiz    old_path   ；章末参照
          make_dir          new_path   ；ディレクトリ NEWDIR を作成
          jc                error_make
          change_dir        new_path   ；カレントディレクトリを
                                       ;NEWDIR に変換
          jc                error_change
          get_dir    2, buffer[03H]    ；カレントディレクトリを得る（47H）
          jc                error_get
          display_asciiz    buffer     ；章末参照
          change_dir        old_path   ；カレントディレクトリの変更（3BH）
          jc                error_change
          rem_dir           new_path   ；ディレクトリ NEWDIR を削除（3AH）
          jc                error_rem
          get_dir    2, buffer[03H]    ；カレントディレクトリを得る（47H）
          jc                error_get
          display_asciiz    buffer     ；章末参照
```

INT 21H

ファンクション 3AH　ディレクトリの削除

コール

AH　　＝3AH
DS：DX＝パス名の位置

リターン

キャリーフラグがセットされた場合

AX ＝03H　無効なパス
　 ＝05H　アクセスの否定（指定されたパスが空でない、あるいはディレクトリでない、またはルートディレクトリであるか、その他の無効な情報が入っている）
　 ＝10H　カレントディレクトリ

キャリーフラグがセットされない場合

エラーなし

解　　説

サブディレクトリを削除します。DX は、削除されるサブディレクトリのパス名を表す ASCIIZ 文字列のオフセットアドレス（DS は、セグメントアドレス）です。削除されるディレクトリは空（ファイル、ディレクトリを含んでいない）でなければなりません。また、カレントディレクトリを削除することはできません。

エラーが起こるとキャリーフラグがセットされ、AX にエラーコードが返されます。

マクロ定義

```
rem_dir     macro   path
            mov     dx, offset path
            mov     ah, 3AH
            int     21H
            endm
```

サンプル

次のプログラムは、ドライブ B のディスク上のルートディレクトリに〝NEWDIR〟という名前のサブディレクトリを作成し、カレントディレクトリを一度〝NEWDIR〟に移してからルートディレクトリに戻り、〝NEWDIR〟を削除します。また、ディレクトリを移動するたびに、カレントディレクトリを表示します。

```
old_path    db        "b:¥", 0, 63 dup(?)
new_path    db        "b:¥newdir", 0
buffer      db        "b:¥", 0, 63 dup(?)
;
func_3AH:   get_dir 2, old_path[03H]
                                   ; カレントディレクトリ情報を得る (47H)
            jc              error_get
            display_asciiz  old_path ; 章末参照
            make_dir        new_path ; ディレクトリ NEWDIR を作成 (39H)
            jc              error_make
            change_dir      new_path ; カレントディレクトリを NEWDIR
                                     ; に変換 (3BH)
            jc              error_change
            get_dir 2, buffer[03H]   ; カレントディレクトリを得る (47H)
            jc              error_get
            display_asciiz  buffer   ; 章末参照
            change_dir      old_path ; カレントディレクトリの変更 (3BH)
            jc              error_change
            rem_dir         new_path ; ディレクトリ NEWDIR を削除 (3AH)
            jc              error_rem
            get_dir 2, buffer[03H]   ; カレントディレクトリを得る (47H)
            jc              error_get
            display_asciiz  buffer   ; 章末参照
```

INT 21H

ファンクション 3BH カレントディレクトリの変更

コール

AH　　=3BH
DS：DX=パス名の位置

リターン

キャリーフラグがセットされた場合
　AX =03H　無効なパス

キャリーフラグがセットされない場合
　エラーなし

解　　説

カレントディレクトリを変更します。DXは、新しいサブディレクトリのパス名を表すASCIIZ文字列のオフセットアドレス（DSは、セグメントアドレス）でなければなりません。ディレクトリを指定する文字列は64文字以内です。

指定されたパス名のディレクトリが存在しないと、カレントディレクトリは変更されません。

エラーが起こるとキャリーフラグがセットされ、AXにエラーコードが返されます。

マクロ定義

```
change_dir  macro   path
            mov     dx, offset path
            mov     ah, 3BH
            int     21H
            endm
```

サンプル

次のプログラムは、ドライブBのディスク上のルートディレクトリに〝NEWDIR〞という名前のサブディレクトリを作成し、カレントディレクトリを一度〝NEWDIR〞に移してからルートディレクトリに戻り、〝NEWDIR〞を削除します。また、ディレクトリを移動するたびに、カレントディレクトリを表示します。

```
old_path    db      "b:¥", 0, 63 dup(?)
new_path    db      "b:¥newdir", 0
buffer      db      "b:¥", 0, 63 dup(?)
;
```

```
func_3BH:   get_dir 2, old_path[03H]
                                    ；カレントディレクトリ情報を得る（47H）
            jc      error_get
            display_asciiz  old_path ；章末参照
            make_dir        new_path ；ディレクトリ NEWDIR を作成（39H）
            jc              error_make
            change_dir      new_path ；カレントディレクトリの変更
            jc              error_change
            get_dir 2, buffer[03H]   ；カレントディレクトリを得る（47H）
            jc              error_get
            display_asciiz  buffer   ；章末参照
            change_dir      old_path ；カレントディレクトリの変更
            jc              error_change
            rem_dir         new_path ；ディレクトリを削除（3AH）
            jc              error_rem
            get_dir 2, buffer[03H]   ；カレントディレクトリを得る（47H）
            jc              error_get
            display_asciiz  buffer   ；章末参照
```

INT 21H

ファンクション 3CH　ハンドルを使うファイルの作成

コール

AH　　＝3CH
DS：DX＝パス名の位置
CX　　＝ファイルの属性

リターン

キャリーフラグがセットされた場合

AX	＝03H	無効なパス
	＝04H	オープンされているファイルが多すぎる（指定された属性のファイルは作成されたが、リード／ライトアクセスをするためのハンドル、または内部システムテーブルに空きスペースがなかった）
	＝05H	アクセスの否定（CXで指定された属性に作成不可能なディレクトリ、ボリュームラベルなどが入っていたか、ファイルを保護する属性がすでに与えられていた。またはディレクトリに同じ名前のファイルが存在していた）

キャリーフラグがセットされない場合

AX ＝ファイルハンドル

解　　説

ファイルを作成し、利用可能な最初のハンドルを割り当てます。DXには新しいファイルのパス名を表すASCIIZ文字列のオフセットアドレス（DSは、セグメントアドレス）を、CXにはファイルに割り当てられた属性を設定します。ファイルの属性については、1.5「ファイルの属性」を参照してください。

同名のファイルが存在しないと、新規のファイルを作成します。同名のファイルがあるときは、そのファイルの大きさを0にします。CX内の属性はファイルに割り当てられ、読み出し／書き込みのためにオープンされます。AXは、ファイルハンドルを返します。

エラーが起こるとキャリーフラグがセットされ、AXにエラーコードが返されます。

マクロ定義

```
create_handle   macro   path, attrib
                mov     dx, offset path
                mov     cx, attrib
                mov     ah, 3CH
                int     21H
```

```
                endm
```

サンプル

次のプログラムは、ドライブBのディスクに〝DIR.TMP〟という名前のファイルを作成します。このファイルは、カレントディレクトリにある各ファイルのファイル名と拡張子を含みます。

```
srch_file   db        "b:*.*", 0
timp_file   db        "b:dir.tmp", 0
buffer      db        43 dup(?)
handle      dw        ?
;
func_3CH:   set_dta buffer          ; ディスク転送アドレスのセット (1AH)
            find_first_file srch_file, 16H ; 最初に一致するファイル名の
                                           ; 検索 (4EH)
            cmp       ax, 12H       ; これ以上ファイルがないか?
            je        all_done      ; はいのとき、all_done へ
            create_handle tmpr_file, 0 ; ハンドルを使うファイルの作成
            jc        error
            mov       handle, ax    ; ハンドルのセーブ
write_it:   write_handle handle, buffer[1EH], 12
                                    ; ファイルを書き込む (40H)
            find_next_file          ; 次に一致するファイル名の検索 (4FH)
            cmp       ax, 12H       ; 他のエントリは存在するか?
            je        all_done      ; いいえのとき、all_done へ
            jmp       write_it      ; はいのとき、レコードを書き込む
all_done:   close_handle handle     ; ハンドルを使うファイルのクローズ (3EH)
```

INT 21H

ファンクション 3DH　ハンドルを使うファイルのオープン

コール

AH　　＝3DH
AL　　＝ファイルアクセスコントロール
DS：DX＝パス名の位置

リターン

キャリーフラグがセットされた場合

AX ＝01H　無効なファンクションコード。またはシェアリングモードが不正なため、ファイルにアクセスできない
　 ＝02H　ファイルが存在しない。またはファイル名が無効
　 ＝03H　パスが存在しない。またはパス名が無効
　 ＝04H　ファイルはこれ以上オープンできない
　 ＝05H　ディレクトリかボリューム ID をオープンしようとした。またはライト不可のファイルに書き込もうとした
　 ＝0CH　アクセスコードが 1、2、3 のいずれでもない

キャリーフラグがセットされない場合

AX ＝ファイルハンドル

解　　説

このファンクションは、システムファイルと隠されたファイルを含む、あらゆるファイルを、入力または出力モードでオープンします。DX にはオープンされるファイルのパス名を表す ASCIIZ 文字列のオフセットアドレス（DS は、セグメントアドレス）を、AL には、ファイルをオープンする方法を表すコード（ファイルアクセスコントロールを参照してください）を設定します。

エラーがないと、MS-DOS は、ハンドルの最初の 1 バイトのリード／ライトの設定をします。

ファイルアクセスコントロール

AL に入れるコードは、次の 3 つのコードの集まりです。

ビット	コード
0〜3	アクセスコード
4〜6	シェアリングモード
7	インヘリッドビット

・アクセスコード

アクセスコード（ALの3〜0ビット）は、ファイルがどのようにアクセスできるかを表します。

ビット3〜0	アクセス	意味
0000	リード	リード不可、リード／ライト不可のシェアリングモードでオープンできません。
0001	ライト	ライト不可、リード／ライト不可のシェアリングモードでオープンできません。
0010	リード／ライト	リード不可、ライト不可、リード／ライト不可のシェアリングモードでオープンできません。

エラーが起こるとキャリーフラグがセットされ、エラーコードがAXに返されます。

・シェアリングモード

シェアリングモード（6〜4ビット）は、他のプロセスが、オープンしているファイルをアクセスできるかどうかを表します。

ファイルを継承する場合、同時にシェアリングモードやアクセスモードも継承します。

ビット6〜4	シェアリングモード	意味
000	コンパチブル	このモードのときは、いかなるプロセスでも、ファイルを何回でもオープンすることができます。他のシェアリングモードのときは、オープンできません。
001	リード／ライト不可	いかなるプロセス（カレントプロセス自身さえも）も、コンパチブルモードでのオープン、読み出し、または書き込みのためのアクセスができません。
011	リード不可	他のプロセスは、コンパチブルモードでのオープン、読み出しのためのアクセスができません。
100	不可なし	他のプロセスは、コンパチブルモードでのオープンができません。

ファイルシェアリングによるエラーのために、システムコールが失敗すると、MS–DOSは割り込みタイプ24H、エラーコード02H（ドライブの準備ができていない）を実行します。続いて実行されるファンクション59H（拡張されたエラーを得る）は、シェアリングの破壊を表す拡張エラーコードを返します。

ファイルをオープンするとき、他のプロセスがこのファイルで実行できる、あらゆる操作の情報をMS–DOSに与えておくことが重要です（シェアリングモード）。デフォルトのシェアリングモード（コンパチブルモード）は、ファイルへの他のプロセスのアクセスをすべて否定します。あるプロセスがファイルを扱っているとき、他のプロセスへそのファイルの読み出しを許可するときは、ビット5をセットしてください。

同様に、カレントプロセスが実行するであろう操作を明確にすることも重要です（アクセスコード）。

デフォルトのアクセスコード（リード／ライト）では、すでにリード不可、ライト不可、リード／ライト不可のいずれかのシェアリングモードでオープンすることはできません。また、あるファイルを読み込むだけの場合、他のすべてのプロセスが、リード不可、リード／ライト不可のどちらかでなければオープンできます。

マクロ定義

```
open_handle macro   path, access
            mov     dx, offset path
            mov     al, access
            mov     ah, 3DH
            int     21H
            endm
```

サンプル

次のプログラムは、ドライブBの〝TEXTFILE.ASC〞という名前のファイルをプリンタに印字します。

```
file        db      "b:textfile, asc", 0
buffer      db      ?
handle      dw      ?
;
func_3DH:   open_handle file, 0         ;ハンドルを使うファイルのオープン
            mov     handle, ax          ;ハンドルのセーブ
read_char:  read_handle handle, buffer, 1  ;1 文字読み込む
            jc      error_read
            cmp     ax, 0               ;ファイルエンドか?
            je      return              ;はいのとき、処理終了
            print_char buffer           ;いいえのとき、文字をプリンタに
                                        ;出力 (05H)
            jmp     read_char           ;次の文字を読み込む
```

INT 21H

ファンクション 3EH ハンドルを使うファイルのクローズ

コール

AH ＝3EH
BX ＝クローズするファイルハンドル

リターン

キャリーフラグがセットされた場合
　AX ＝ 06H　無効なハンドル（オープンされていないハンドル）

キャリーフラグがセットされない場合
　エラーなし

解説

ファンクション 3DH（ハンドルを使うファイルのオープン）、または 3CH（ハンドルを使うファイルの作成）によって、オープンされたファイルをクローズします。

エラーがないと、MS-DOS はファイルをクローズし、すべての内部バッファを開放します。エラーがおこるとキャリーフラグがセットされ、AX にエラーコードが返されます。

マクロ定義

```
close_handle    macro   handle
                mov     bx, handle
                mov     ah, 3EH
                int     21H
                endm
```

サンプル

次のプログラムは、ドライブ B のディスク上に〝DIR.TMP〞という名前のファイルを作ります。このファイルは、カレントディレクトリにあるファイルのファイル名と拡張子を含んでいます。

```
srch_file   db      "b:*.*", 0
tmp_file    db      "b:dir.tmp", 0
buffer      db      43 dup(?)
handle      dw      ?
;
func_3EH:   set_dta buffer       ;ディスク転送アドレスのセット (1AH)
```

```
            find_first_file srch_file, 16H
                                ; 最初に一致するファイル名の検索 (4EH)
            cmp     ax, 12H     ; これ以上ファイルがないか?
            je      all_done    ; はいのとき、all_done へ
            create_handle  tmp_file, 0  ; ハンドルを使うファイルの作成 (3CH)
            jc      error_create
            mov     handle, ax  ; ハンドルのセーブ
write_it:   write_handle handle, buffer[1EH], 12
                                ; ファイルを書き込む (40H)
            jc      error_write
            fine_next_file      ; 次に一致するファイル名の検索 (4FH)
            cmp     ax, 12H     ; 他のエントリは存在するか?
            je      all_done    ; いいえのとき、all_done へ
            jmp     write_it    ; はいのとき、レコードを書き込む
all_done:   close_handle handle ; ハンドルを使うファイルのクローズ (3EH)
            jc      error_close
```

INT 21H

ファンクション
3FH ファイルかデバイスの読み出し

コール

AH　　＝3FH
DS：DX＝バッファの位置
CX　　＝読み込むべきバイト数
BX　　＝ファイルハンドル

リターン

キャリーフラグがセットされた場合
　AX ＝05H　アクセスできない（ハンドルがリード許可されていない）
　　＝06H　ハンドルが無効（ハンドルがオープンされていない）

キャリーフラグがセットされない場合
　AX ＝読み出されたバイト数

解説

ハンドルで指定されたファイル、またはデバイスからデータを読み出します。BXにはハンドル、CXには読み出すバイト数、DXにはバッファのオフセットアドレス（DXは、セグメントアドレス）を設定します。

エラーがないと、AXは読み出されたバイト数を返します。ファイルの先頭がEOF（ファイルの終りを表すコード）のとき、AXは0を返します。CXで指定されたバイト数が、すべてバッファに転送される保証はありません。たとえば、このファンクションを使ってキーボードからデータを読み出すとき、最高1行分（最初のキャリッジリターンを入力するまで）のデータしか読み出しません。

このファンクションを使って標準入力から読み出しを行うと、リダイレクト処理が可能になります。

マクロ定義

```
read_handle macro   handle, buffer, bytes
            mov     bx, handle
            mov     dx, offset buffer
            mov     cx, bytes
            mov     ah, 3FH
            int     21H
            endm
```

サンプル

次のプログラムは、ドライブBのディスク上の〝TEXTFILE.ASC〟という名前のファイルを表示します。

```
filename    db      "b:¥textfile.asc", 0
buffer      db      129 dup(?)
handle      dw      ?
;
func_3FH:   open_handle filename, 0
                         ; ハンドルを使うファイルのオープン（3DH）
            jc      error_open
            mov     handle, ax          ; ハンドルのセーブ
read_file:  read_handle buffer, file_handle, 128
            jc      error_open
            cmp     ax, 0               ; ファイルエンドか?
            je      return              ; はいのとき、処理終了
            mov     bx, ax              ; 読み出したバイト数を Bx にセット
            mov     buffera[bx], "$"    ; 表示する文字列を作成
            display buffer              ;buffer を画面に表示（09H）
            jmp     read_file           ; 続けて読み出す
```

INT 21H

ファンクション 40H ファイルかデバイスへの書き込み

コール

AH ＝40H
DS：DX＝バッファの位置
CX ＝書き込むべきバイト数
BX ＝ファイルハンドル

リターン

キャリーフラグがセットされた場合
AX ＝05H アクセスの否定（ハンドルがリード許可されていない）
＝06H 無効なハンドル（ハンドルがオープンされていない）

キャリーフラグがセットされない場合
AX ＝書き込まれたバイト数

解説

ハンドルで指定されたファイル、またはデバイスにデータを書き込みます。BX にはハンドル、CX には書き込むバイト数、DX には書き込むデータのオフセットアドレス（DX は、セグメントアドレス）を設定します。

エラーがないと、AX は書き込まれたバイト数を返します。ディスクにファイルを書き込んだ後は、必ず AX をチェックしてください。AX が 0 であると、ディスクに書き込む余裕がないことを表します。このコールが実行された後で、AX の値が CX で指定された値より少ないと、キャリーフラグはセットされませんが、エラーであることを表します。

標準出力に書き込んだ場合、出力はリダイレクト可能になります。このファンクションで、CX = 0（バイト数= 0）を指定すると、ファイルサイズは現在のリード／ライトポインタの値にセットされます。クラスタは、新しいファイルのサイズを満たすように割り付け、または開放されます。

エラーが起こるとキャリーフラグがセットされ、AX にエラーコードが返されます。

マクロ定義

```
write_handle    macro   handle, data, bytes
                mov     bx, handle
                mov     dx, offset data
                mov     cx, bytes
                mov     ah, 40H
                int     21H
```

```
          endm
```

サンプル

次のプログラムは、ドライブＢのディスクに〝DIR.TMP〟という名前のファイルを作成します。このファイルは、カレントディレクトリにある、各ファイルのファイル名と拡張子を含んでいます。

```
srch_file  db      "b:*.*", 0
tmp_file   db      "b:dir.tmp", 0
buffer     db      43 dup(?)
handle     dw      ?
;
func_40H:  set_dta buffer  ；ディスク転送アドレスのセット（1AH）
           find_first_file  srch_file, 16H ；最初に一致するファイル名の検索（4EH）
           cmp     ax, 12H           ；これ以上ファイルがないか?
           je      return            ；はいのとき、処理終了
           create_handle tmp_file, 0
                         ；ハンドルを使うファイルの作成（3CH）
           jc      error_create
           mov     handle, ax        ；ハンドルのセーブ
wtite_it:  write_handle  handle, buffer[1EH], 12  ；ファイルに書き込む
           jc      error_write
           find_next_file            ；次の一致するファイル名の検索（4FH）
           cmp     ax, 12H           ；他のエントリは存在するか?
           je      all_done          ；いいえのとき、処理終了
           jmp     write_it          ；はいのとき、レコードを書き込む
all_done:  close_handle  handle
                         ；ハンドルを使うファイルのクローズ（3EH）
           jc      error_close       ；エラー処理
```

INT 21H

ファンクション 41H ディレクトリエントリの削除

コール

AH　　＝41H
DS：DX＝パス名の位置

リターン

キャリーフラグがセットされた場合

AX ＝02H　無効なファイル（指定されたファイルが存在しない）
　　＝03H　無効なパス（指定されたパスが存在しない）
　　＝05H　アクセスの否定（指定されたパスがディレクトリ、またはリードオンリーのファイルであった）

キャリーフラグがセットされない場合

エラーなし

解　　説

ディレクトリエントリを削除することによって、ファイルを削除します。DX は、削除するファイルのパス名を表す ASCIIZ 文字列のオフセットアドレス（DS は、セグメントアドレス）でなければなりません。ワイルドカード文字は使用できません。

ファイルが存在し、読み出し専用のファイルでなければファイルを削除します。エラーが起こるとキャリーフラグがセットされ、AX にエラーコードが返されます。

属性が読み出し専用のファイルを削除するときは、ファンクション 43H（属性の変更）によって属性を変更してください。

マクロ定義

```
delete_entry    macro   path
                mov     dx, offset path
                mov     ah, 41H
                int     21H
                endm
```

サンプル

次のプログラムは、ドライブ B のディスク上の 1990 年 12 月 31 日以前の日付のファイルをすべて消去します。

```
year      db      1990
month     db      12
day       db      31
files     db      ?
message   db      "NO FILES DELETED.", ODH, OAH, "$"
path      db      "b:*.*", 0
buffer    db      43dup(?)
;
func_41H: set=dta buffer                  ; ディスク転送アドレスのセット (1AH)
          select_disk  "B"                ; ドライブ B を選択 (OEH)
          find_first_file path, 0
                                  ; 最初に一致するファイル名の検索 (4EH)
          jnc     compare                 ; 一致するファイルを得る
          jmp     all_done                ; 一致しないとき、all_done へ
compare:  convert_date     buffer[-1]     ; 章末参照
          cmp     cx, year                ; 年は 1990 より大きいか?
          jg      next                    ; はいのとき、ファイルを削除しない
          cmp     dl, month               ;12 月を超えているか?
          jg      next                    ; はいのとき、ファイルを削除しない
          cmp     dh, day                 ;31 日以上か?
          jge     next                    ; はいのとき、削除しない
          delete_entry  buffer[1EH]       ; ディレクトリエントリの削除
          jc      error_delete
          inc     files                   ; ファイルカウンタをインクリメント
next:     find_next_file                  ; 次に一致するファイルの検索
          jnc     compare                 ; 日付チェック処理を継続
how_many: cmp     tiles, 0                ; これ以上ファイルがないか?
          je      all_done                ; はいのとき、all_done へ
          convert files, 10, message      ; 章末参照
          all_done  display  message      ;message を画面に表示 (09H)
          select_disk "A"                 ; ドライブ A を選択 (OEH)
```

INT 21H

ファンクション
42H ファイルポインタの移動

コール

AH　　＝42H
CX：DX＝移動するバイト数
AL　　＝移動方法（解説参照）
BX　　＝ファイルハンドル

リターン

キャリーフラグがセットされた場合
　AX ＝01H　無効なファンクション
　　 ＝06H　オープンされていないハンドルを指定した

キャリーフラグがセットされない場合
　DX：AX＝新規のポインタロケーション

解説

ハンドルで指定されるファイルのリード／ライトポインタを移動します。BX にはハンドル、CX：DX には 32 ビットのオフセット（CX が上位 16 ビット、DX が下位 16 ビットを表します）を設定します。AL はポインタの移動方法で、次の値で指定します。

AL の値	機　能
00H	ポインタは、ファイルの先頭からオフセットの位置に移動する。
01H	ポインタは、現在のロケーション（アドレス）とオフセットを加算した位置に移動する。
02H	ポインタは、ファイルの終わりにオフセットを加算した位置に移動する。

DX：AX は、新規のリード／ライトポインタロケーション（32 ビット整数：DX が上位 16 ビット、AX が下位 16 ビットを表します）を返します。CX：DX を 0、AL を 2 にして、このファンクションをコールし、ファイルの大きさを設定できます。このとき、DX：AX は、ファイルの大きさ（ファイルの最後のバイトの次のバイトのオフセット）をバイトで返します。

マクロ定義

```
move_ptr   macro   handle, high, low, method
           mov     bx, handle
           mov     cx, high
```

```
        mov     dx, low
        mov     al, method
        mov     ah, 42H
        int     21H
        endm
```

サンプル

次のプログラムは、1文字の入力を要求し、それを対応する数字に変換（A = 01H、B = 02H、…）します。次に、その数字番目のレコード内容をファイルから読み出して表示します。読み出すファイルは、ドライブBのカレントディレクトリにある"ALPHABET.DAT"で、1レコード28バイトで26レコードからなります。

```
file          db      "b:alphabet.dat", 0
buffer        db      28 dup(?), "$"
prompt        db      "Enter letter:$"
crlf          db      0DH, 0AH, "$"
handle        db      ?
record_length  dw   28
;
func_42H:   open_handle  file, 0  ; ハンドルを使うファイルのオープン (3DH)
            jc      error_open
            mov     handle, ax              ; ハンドルをセーブ
get_char:   display prompt                  ;prompt を画面に表示 (09H)
            read_kbd_and_echo               ;1 文字の入力待ち (01H)
            sub     al, 41h                 ; 入力文字をレコード番号に変換
            mul     byte ptr record_length  ; オフセットを算出
            move_ptr  handle, 0, ax, 0      ; ファイルポインタを移動
            jc      error_move
            read_handle  handle, buffer, record_length
            jc      error_read
            cmp     ax, 0                   ; ファイルエンドか?
            je      return                  ; はいのとき、処理終了
            display crlf                    ;crlf を画面に出力 (09H)
            display buffer                  ;buffer を画面に表示 (09H)
            display crlf                    ;crlf を画面に出力 (09H)
            jmp     get_char                ; 次の文字を得る
```

INT 21H

ファンクション 43H ファイルの属性の取得／設定

コール

AH　　＝43H
DS：DX＝パス名の位置
CX　　＝（AL＝01H の場合）　セットする属性
AL　　＝00H　ファイルの現在の属性を返す
　　　＝01H　CX で指定された属性の設定

リターン

キャリーフラグがセットされた場合
AX ＝01H　無効なファンクション
　 ＝02H　ファイルが存在しない。またはファイル名が無効
　 ＝03H　パスが存在しない。またはパス名が無効
　 ＝05H　ディレクトリかボリューム ID にアクセスしようとした

キャリーフラグがセットされない場合
CX ＝属性（AL＝00H の場合）

解　説

ファイルの属性を取得、または設定します。DX にはファイルのパス名を表す ASCIIZ 文字列のオフセットアドレス（DS は、セグメントアドレス）、AL には属性を取得するか設定するかを決めるパラメータ（0：属性を取得する、1：属性を設定する）を指定します。

AL が 0 のとき（属性を取得する）、属性を表す 1 バイトが CX に返されます。AL が 1 のとき（属性を設定する）、CX にはセットする属性を設定します。属性については 1.5「ファイルの属性」を参照してください。

このファンクションを使って、属性のボリューム ID ビット（08H）、またはディレクトリビット（10H）を変更することはできません。

エラーが起こるとキャリーフラグがセットされ、AX にエラーコードが返されます。

マクロ定義

```
change_attr macro   path, action, attrib
            mov     dx, offset path
            mov     al, action
            mov     cx, attrib
            mov     ah, 43H
```

```
          int     21H
          endm
```

サンプル

次のプログラムは、ドライブBにあるディスクの、カレントディレクトリにある〝REPORT.ASM〟というファイルの属性を表示します。

```
header      db      15 dup(20h), "Read-", 0DH, 0AH
            db      "Filename Only Hidden"
            db      "System Volume Sub-Dir Archive"
            db      0DH, 0AH, 0DH, 0AH, "$"
path        db      "b:report.asm", 3 dup(0), "$"
attribute   dw      ?
blanks      db      9 dup(20h), "$"
;
func_43H:   change_attr path, 0, 0   ; 属性を得る
            jc      error_mode
            mov     attribute, cx    ; 属性をセーブ
            display header           ;header を画面に表示 (09H)
            display path             ;path を画面に表示 (09H)
            mov     cx, 6            ; (0～5) の6ビットをチェック
            mov     bx, 1
chk_bit:    test    attribute, bx    ; ビットがセットされているか?
            jz      no_attr          ; いいえのとき、no_attr へ
            display_char  "X"        ; はいのとき、"x"を画面に出力 (02H)
            jmp short next_bit       ; 次のビット処理へ
no_attr:    display_char 20h         ; 空白を画面に出力 (02H)
next_bit:   display blanks           ;blanks を画面に表示 (09H)
            shl     bx, 1            ; 次のビットにシフト
            loop    chk_bit          ; それをチェック
```

INT 21H

IOCTL データの取得

コール

AH ＝44H
AL ＝00H
BX ＝ハンドル

リターン

キャリーフラグがセットされた場合
AX ＝01H　無効なファンクション（AL が 00H でない）
　＝06H　無効なハンドル（ハンドルがオープンされていない）

キャリーフラグがセットされない場合
DX ＝デバイスデータ

解　説

デバイスコントロールデータを得ます。AL には 00H、BX にはハンドルを設定します。

2 バイトのデバイスデータは DX に返されます。デバイスデータのビット 7 によって、ハンドルがファイルを表すかデバイスを表すかが決まり、他のビットの意味も異なります。

・デバイス（ビット 7＝ 1）の場合

ビット	値	意　味
15		予備
14	1	この装置はファックション 4402H（IOCTL キャラクタを受け取る）と 03H（IOCTL キャラクタを送る）を通して、コントロール文字列を処理できる。このビットは読み出すことはできるが、書き込むことはできない
8〜13		予備
7	1	ハンドルはデバイスを表す
6	0	EOF を入力する場合
5	1	コントロールキャラクタをチェックしない
	0	コントロールキャラクタをチェックする
4	1	予備
3	1	クロックデバイス
2	1	NUL デバイス
1	1	コンソール出力
0	1	コンソール入力

ビット5がチェックできるコントロールキャラクタは、<CTRL–C>、<CTRL–P>、<CTRL–S>、<CTRL–Z>で、データとして扱うかコントロールキャラクタとして扱うかを決めます。ビット5をセットし、<CTRL–C>をデータとして扱う場合、ファンクション33H（<CTRL–C>チェックのセット／リセット）またはMS–DOSのBREAKコマンドで、<CTRL–C>をチェックしないようにしなければなりません。

・ファイル（ビット7＝0）の場合

ビット	値	意　味
15〜8		予備
7	0	ハンドルはファイルを表す
6	0	書き込まれたファイル
5〜0		デバイス番号（00H ＝ A、01H ＝ B、…）

エラーが起こるとキャリーフラグがセットされ、AXにエラーコードを返します。

マクロ定義

```
ioctl_data  macro    code, handle
            mov      bx, handle
            mov      al, code
            mov      ah, 44H
            int      21H
            endm
```

サンプル

次のプログラムは標準出力のデバイスデータを得て、コントロールキャラクタをチェックしないようにビット5をセットし、次にビット5を0にします。

```
get         equ     0
set         equ     1
stdout      equ     1
;
func_4400H: ioctl_data  get, stdout ;IOCTL データを得る
            jc      error
            mov     dh, 0            ;DH をクリア
            or      dl, 20H          ; ビットをセット
            ioctl_data  set, stdout ;IOCTL データをセット
            jc      error
;
; コントロールキャラクタは、ここではデータとして扱う（"raw mode"）
;
            ioctl_data  get, stdout ;IOCTL データを得る
            jc      error
```

```
                mov         dh, 0           ;DH をクリア
                and         dl, 0DFH        ; ビット 5 をクリア
                ioctl_data  set, stdout     ;IOCTL データをセット
;
; コントロールキャラクタは、ここでは処理される（"cooked mode"）
;
```

INT 21H

ファンクション
4401H IOCTLデータの設定

コール

AH ＝44H
AL ＝01H
BX ＝ハンドル
DX ＝デバイスデータ（DH＝0）

リターン

キャリーフラグがセットされた場合

AX ＝01H　無効なファンクション（ALが01HでないかALが01HでDHが00Hでない）
＝06H　無効なハンドル（ハンドルがオープンされていない）

キャリーフラグがセットされない場合

DX ＝デバイスデータ

解　説

デバイスコントロールデータをセットします。ALには01H、BXにはハンドル、DHには00Hを設定します。

デバイスデータの2バイトは、DXの内容にセットされます。デバイスデータのビット7によって、ハンドルがファイルを表すかデバイスを表すかが決まり、他のビットの意味も異なります。

・デバイス（ビット7＝1）の場合

ビット	値	意　味
15		予備
14	1	この装置はファンクション4402H（IOCTLキャラクタを送る）と03H（IOCTLキャラクタを受け取る）を通して、コントロール文字列を処理できる。このビットは読み出すことはできるが、書き込みはできない
13〜8		予備
7	1	ハンドルはデバイスを表す
6	0	EOFを入力する
5	1	コントロールキャラクタをチェックしない
	0	コントロールキャラクタをチェックする

ビット	値	意　味
4	1	予備
3	1	クロックデバイス
2	1	NUL デバイス
1	1	コンソール出力
0	1	コンソール入力

ビット5がチェックできるコントロールキャラクタは、<CTRL–C>、<CTRL–P>、<CTRL–S>、<CTRL–Z>で、データとして扱うかコントロールキャラクタとして扱うかを決めます。ビット5をセットして<CTRL–C>をデータとして扱う場合、ファンクション33H（<CTRL–C>チェックのセット／リセット）またはMS–DOSのBREAKコマンドで、<CTRL–C>をチェックしないようにしなければなりません。

・ファイル（ビット7＝0）の場合

ビット	値	意　味
15〜8		予備
7	0	ハンドルはファイルを表す
6	0	書き込まれたファイル
5〜0		デバイス番号（00H＝A：、01H＝B：、…）

エラーが起こるとキャリーフラグがセットされ、AXにエラーコードを返します。

マクロ定義

```
ioctl_data  macro   code, handle
            mov     bx, handle
            mov     al, code
            mov     ah, 44H
            int     21H
            endm
```

サンプル

ファンクション4400Hを参照してください。

INT 21H

ファンクション 4402H　IOCTLキャラクタを受け取る

コール

AH　　=44H
AL　　=02H
BX　　=ハンドル
CX　　=コントロールデータのバイト数
DS：DX=バッファのアドレス

リターン

キャリーフラグがセットされた場合
AX =01H　無効なファンクション（ALが02Hでないか、デバイスがファンクションに適合しない）
　 =06H　無効なハンドル（ハンドルがオープンされていない）

キャリーフラグがセットされない場合
AX =転送されたバイト数

解　　説

コントロールデータを、キャラクタデバイスから受け取ります。ALは02Hでなければなりません。BXは、プリンタやシリアルポートのようなキャラクタデバイスのハンドルでなければなりません。CXは、読み取るコントロールデータのバイト数です。DXは、データバッファのオフセットアドレスです（DSは、セグメントアドレス）。

AXは、転送されたバイト数を返します。デバイスドライバは、IOCTLインターフェイスをサポートしているものでなければなりません。

エラーが起こるとキャリーフラグがセットされ、AXにエラーコードを返します。

マクロ定義

```
ioctl_char  macro   code, handle, buffer
            mov     bx, handle
            mov     dx, offset buffer
            mov     al, code
            mov     ah, 44H
            int     21H
            endm
```

サンプル

ファンクション4400Hを参照してください。

INT 21H

IOCTLキャラクタを送る

コール

AH =44H
AL =03H
BX =ハンドル
CX =コントロールデータのバイト数
DS：DX=バッファのアドレス

リターン

キャリーフラグがセットされた場合
AX =01H 無効なファンクション（ALが03Hでないか、デバイスがファンクションに適合しない）
=06H 無効なハンドル（ハンドルがオープンされていない）
キャリーフラグがセットされない場合
AX =転送されたバイト数

解説

IOCTLコントロールデータをキャラクタデバイスに送ります。ALは03Hでなければなりません。BXは、プリンタやシリアルポートのようなキャラクタデバイスのハンドルです。CXは、書き込むべきコントロールデータのバイト数です。DXは、データバッファのオフセットアドレスです（DSは、セグメントアドレス）。

AXは、転送されたバイト数を返します。デバイスドライバは、IOCTLインターフェイスをサポートしているものでなければなりません。

エラーが起こるとキャリーフラグがセットされ、AXにエラーコードを返します。

マクロ定義

```
ioctl_char  macro   code, handle, buffer
            mov     bx, handle
            mov     dx, offset buffer
            mov     al, code
            mov     ah, 44H
            int     21H
            endm
```

サンプル

ファンクション4400Hを参照してください。

INT 21H

ファンクション 4404H　IOCTLブロックを受け取る

コール

AH　　=44H
AL　　=04H
BL　　=ドライブ番号（00H　=カレント、01H　=A：、…）
CX　　=コントロールデータのバイト数
DS：DX=バッファのアドレス

リターン

キャリーフラグがセットされた場合
AX =01H　無効なファンクション（AL が 04H でないか、デバイスがファンクションに適合しない）
　=05H　無効なドライブ番号

キャリーフラグがセットされない場合
AX =転送されたバイト数

解　説

コントロールデータをブロックデバイスから受け取ります。AL は 04H でなければなりません。BL はドライブ番号（00H =カレント、01H = A、…）、CX は転送されるべきコントロールデータのバイト数です。DX はデータバッファのオフセットアドレスです（DS は、セグメントアドレス）。

AX は、転送されたバイト数を返します。デバイスドライバは、IOCTL インターフェイスをサポートしているものでなければなりません。ファンクション 4400H 実行の結果、ビット 14 が 1 であると、そのドライバは IOCTL をサポートしています。

エラーが起こるとキャリーフラグがセットされ、AX にエラーコードを返します。

マクロ定義

```
ioctl_block macro    code, drive, buffer
            mov      bl, drive
            mov      dx, offset buffer
            mov      al, code
            mov      ah, 44H
            int      21H
            endm
```

サンプル

ファンクション 4400H を参照してください。

INT 21H

IOCTL ブロックを送る

コール

AH ＝44H
AL ＝05H
BL ＝ドライブ番号（00H ＝カレント、01H ＝A：、…）
CX ＝コントロールデータのバイト数
DS：DX＝バッファのアドレス

リターン

キャリーフラグがセットされた場合
AX ＝01H 無効なファンクション（AL が 05H でないか、デバイスがファンクションに適合しない）
＝05H 無効なドライブ番号

キャリーフラグがセットされない場合
AX ＝転送されたバイト数

解　　説

コントロールデータをブロックデバイスに送ります。AL は 05H でなければなりません。BL はドライブ番号（00H＝カレント、01H＝A、…）、CX は転送されるべきコントロールデータのバイト数です。DX は、データバッファのオフセットアドレスです（DS は、セグメントアドレス）。

AX は、転送されたバイト数を返します。デバイスドライバは、IOCTL インターフェイスをサポートしているものでなければなりません。ファンクション 4400H 実行の結果、ビット 14 が 1 であれば、そのドライバは IOCTL をサポートしています。

エラーが起こるとキャリーフラグがセットされ、AX にエラーコードを返します。

マクロ定義

```
ioctl_block macro   code, drive, buffer
            mov     bl, drive
            mov     dx, offset buffer
            mov     al, code
            mov     ah, 44H
            int     21H
            endm
```

サンプル

ファンクション 4400H を参照してください。

INT 21H

ファンクション 4406H 入力ステータスのチェック

コール

AH ＝ 44H
AL ＝ 06H
BX ＝ハンドル

リターン

キャリーフラグがセットされた場合

AX ＝ 01H　無効なファンクション（AL の値が不正）
　＝ 05H　アクセスが否定された
　＝ 06H　無効なハンドルを指定した。またはハンドルがオープンされている
　＝ 0DH　無効なデータ

キャリーフラグがセットされない場合

AL ＝ 00H　レディ状態でない
　＝ FFH　レディ

解　説

ハンドルがレディ状態かどうかをチェックします。AL は、06H でなければなりません。BX はハンドルです。AL の返す値とステータスの関係は次のとおりです。

値	デバイスのときの意味	入力ファイルのときの意味
00H	レディ状態ではない	ポインタが EOF を指している
FFH	レディ状態	レディ状態

マクロ定義

```
ioctl_status    macro   code, handle
                mov     bx, handle
                mov     al, code
                mov     ah, 44H
                int     21H
                endm
```

サンプル

次のプログラムは、ハンドルの入力ステータスが、レディ状態かポインタがEOFを指しているかを表示します。

```
stdin       equ     0
stdout      equ     1
;
message     db      "File is"
ready       db      "ready."
at_eof      db      "at EOF."
crlf        db      0DH, 0AH
;
func_4406H: write_handle  stdout, message, 8 ;message を表示
            jc      write_error
            ioctl_status  6, stdin          ; 入力ステータスをチェック
            jc      ioctl_error
            cmp     al, 0                   ; 入力ステータスはレディか?
            jne     not_eof                 ; はいのとき、not_eof へ
            write_handle  stdout, at_eof, 7 ;at_eof を表示 (40H)
            jc      write_error
            jmp     all_done                ;all_done へ
not_eofr:   write_handle  stdout, raedy, 6  ;ready を表示 (40H)
all_done:   write_handle  stdout, crlf, 2   ;crlf を表示 (40H)
            jc      write_error             ; エラー処理へ
```

INT 21H

ファンクション 4407H　出力ステータスのチェック

コール

AH ＝44H
AL ＝07H
BX ＝ハンドル

リターン

キャリーフラグがセットされた場合

AX ＝01H 無効なファンクション（ALの値が不正）
　＝05H アクセスが否定された
　＝06H 無効なハンドル
　＝0DH 無効なデータ

キャリーフラグがセットされていない場合

AL ＝00H レディ状態ではない
　＝FFH レディ状態である

解　説

ハンドルがレディ状態かどうかをチェックします。ALは、07Hでなければなりません。BXはハンドルです。ALの返す値とステータスの関係は次のとおりです。

値	デバイスのときの意味	出力ファイルのときの意味
00H	レディ状態ではない	レディ状態
FFH	レディ状態	レディ状態

出力ファイルは、たとえディスクがfullになっても、レディ状態を返します。エラーが起こるとキャリーフラグがセットされ、AXにエラーコードを返します。

マクロ定義

```
ioctl_status    macro   code, handle
                mov     bx, handle
                mov     al, code
                mov     ah, 44H
                int     21H
                endm
```

サンプル

ファンクション4406Hを参照してください。

INT 21H

ファンクション 4408H　IOCTL：媒体が交換可能か調べる

コール

AH ＝44H
AL ＝08H
BL ＝ドライブ番号（00H ＝カレント、01H ＝A：、…）

リターン

キャリーフラグがセットされた場合

AX ＝01H　無効なファンクション（ALの値が不正か、デバイスがサポートされていない）
＝0FH　無効なドライブ番号

キャリーフラグがセットされていない場合

AX ＝00H　交換可能
＝01H　交換不可能

解　　説

指定したドライブの記憶媒体が交換可能なものか不可能なものかを調べます。正常にリターンしたとき、AXが01Hであると固定ディスクのように交換不可能なドライブ、AXが00Hであると通常のディスクのように交換可能なドライブです。

このファンクションが実行されると、ディスクを交換するか否かのメッセージが出されます。

エラーが起こるとキャリーフラグがセットされ、AXにエラーコードを返します。

マクロ定義

```
ioctl_change    macro   drive
                mov     bl, drive
                mov     ah, 08H
                mov     ah, 44H
                int     21H
                endm
```

サンプル

次のプログラムは、カレントディスクが交換できるかどうかを調べ、交換できないディスクの場合は作業を続け、交換できる場合はディスクを差し換える旨のメッセージを出します。

```
stdout       equ     1
;
message      db      "Please replace disk in drive"
drives       db      "ABCD"
crlf         db      0DH, 0AH
;
func_4408H: ioctl_change    0           ;IOCTL の交換性のチェック
            jc      ioctl_error
            cmp     ax, 0               ; カレントドライブの交換は可能か?
            jne     next_process        ; いいえのとき、次の処理へ
            write_handle  stdout, message, 29  ; はいのとき、
                                                ;message を表示 (40H)
            jc      write_error
            current_disk                ; カレントドライブ番号を得る (19H)
            xor     bx, bx              ; インデックスをクリア
            mov     bl, al              ; カレントドライブ番号をセット
            display_char  drives[bx]    ; カレントドライブを画面
                                        ; に出力 (02H)
            write_handle  stdout, crlf, 2  ;crlf を表示 (40H)
            jc      write_error
next_process:
;
;          (further processing here)
```

INT 21H

IOCTL：リモートブロックデバイスの検出

コール

AH ＝44H
AL ＝09H
BL ＝ドライブ番号（00H＝カレント、01H＝A：、…）

リターン

キャリーフラグがセットされた場合

AX ＝01H　無効なファンクション（ALの値が不正か、SHARE.EXEが常駐していない）
　 ＝0FH　無効なドライブ番号

キャリーフラグがセットされていない場合

DX ＝デバイス属性ワード

解　説

このファンクションは、ドライブ名がMS–Networksのワークステーション（ローカル）のドライブであるか、サーバ（リモート）へリディレクトされているかをチェックします。BLは、ドライブ番号（00H＝カレント、01H＝A：、…）です。

ブロックデバイスがローカルであると、DXはデバイスヘッダの属性ワード（2バイト）を返します。ブロックデバイスがリモートであると、ビット12だけがセットされ（1000H）、他のビットは0（予備）になります。

アプリケーションプログラムから、ビット12をチェックすることはできません。したがって、ローカル、リモート、デバイスの区別ができません。

エラーが起こるとキャリーフラグがセットされ、AXにエラーコードを返します。

マクロ定義

```
ioctl_rblock    macro   drive
                mov     bl, drive
                mov     al, 09H
                mov     ah, 44H
                int     21H
                endm
```

サンプル

次のプログラムは、ドライブBがローカルかリモートかをチェックし、適切なメッセージを表示します。

```
stdout      equ         1
;
message     db          "Drive B: is"
loc         db          "local."
rem         db          "remote."
crlf        db          0DH, 0AH
;
func_4409H: write_handle  stdout, message, 12  ;message を表示
            jc       write_error
            ioctl_rblock  2               ;ドライブ B がローカルかリモート
                                          ;かをチェック
            jc       ioctl_error
            test     dx, 1000h       ;ビット 12 がセットされているか?
            jnz      not_loc         ;はいのとき、リモートでる、not_loc へ
            write_handle  stdout, loc, 6
                                     ;loc を表示 (40H)
            jc       write_error
            jmp      done
not_loc:    write_handle  stdout, rem, 7
                                     ;rem を表示 (40H)
            jc       write_error
done:       write_handle  stdout, crlf, 2
                                     ;crlf を表示 (40H)
            jc       write_error
```

INT 21H

IOCTL：リモートハンドルの検出

コール

AH ＝44H
AL ＝0AH
BX ＝ハンドル

リターン

キャリーフラグがセットされた場合

AX ＝01H　無効なファンクションコード（ALの値が不正か、MS-Networksが稼働していない）
　　＝06H　無効なハンドル

キャリーフラグがセットされていない場合

DX ＝IOCTLビットフィールド

解　　説

このファンクションは、ファイルがMS-Networksのワークステーション（ローカル）のファイルまたはディスクであるか、サーバへリディレクトされているかをチェックします。BXはファイルハンドルです。DXはIOCTLビットフィールドを返します。ビット15が1であると、ハンドルはリモートファイルかディスクです。

アプリケーションプログラムから、ビット15をチェックすることはできません。したがって、アプリケーションプログラムはローカル／リモートを区別するべきではありません

エラーが起こるとキャリーフラグがセットされ、AXにエラーコードを返します。

マクロ定義

```
ioctl_rhandle    macro   handle
                 mov     bx, handle
                 mov     al, 0AH
                 mov     ah, 44H
                 int     21H
                 endm
```

サンプル

次のプログラムは、ハンドル5がローカルか、リモートのファイルか、デバイスかを表示します。

```
stdout      equ         1
;
message     db          "Handle 5 is"
loc         db          "local."
rem         db          "remote."
crlf        db          0DH, 0AH
;
func_440AH: write_handle  stdout, message, 12   ;message を表示
            jc      write_error
            ioctl_rhandle 5                 ; ハンドル 5 がローカルかリモート
                                            ; かをチェック
            jc      ioctl_error
            test    dx, 8000h               ; ビット 15 がセットされているか?
            jnz     not_loc                 ; はいのとき、リモートである、
                                            ;not_loc へ
            write handle  stdout loc, 6 ;loc を表示 (40H)
            jc      write_error
            jmp     done
not_loc:    write_handle  stdout, rem, 7 ;rem を表示 (40H)
            jc      write_error
done:       write_handle  stdout, crlf, 2  ;crlf を表示 (40H)
            jc      write_error
```

INT 21H

IOCTL：リトライ回数の変更

コール

AH ＝ 44H
AL ＝ 0BH
DX ＝リトライの回数
CX ＝待ち時間

リターン

キャリーフラグがセットされた場合

AX ＝ 01H　無効なファンクション（AL の値が不正か、SHARE.EXE が常駐していない）

キャリーフラグがセットされない場合

エラーなし

解　説

このファンクションは、ファイルの共有違反が発生したとき、MS-DOS が行うリトライの回数をセットします。DX にはリトライの回数、CX にはリトライする間隔を時間で指定します。

MS-DOS は、このファンクションによって変更されない限り、リトライを 3 回行います。指定されたリトライをセットした後、要求されたプロセスのために、MS-DOS は割り込みタイプ 24H を実行します。

CX で与えた待ち時間に対し、実際に必要な時間は機種によって異なります。これは、MS-DOS が用意した待ち時間のループ、CPU の処理速度とクロックサイクルに依存します。ユーザーが実際の時間を知っていて、それをもとに設定したい場合、リトライの回数を 1 にして、待ち時間をいろいろ変えてみてください。

エラーが起こるとキャリーフラグがセットされ、AX にエラーコードを返します。

注意　このシステムコールを使うには、ファイルシェアリング（SHARE.EXE）のロード（常駐）が必要です

マクロ定義

```
ioctl_retry macro   retries, wait
            mov     dx, retries
            mov     cx, wait
            mov     al, 0BH
            mov     ah, 44H
```

```
            int      21H
            endm
```

サンプル

次のプログラムはリトライの回数を 10 にし、待ち時間を 1000 にします。

```
func_440BH: ioctl_retry 10, 1000     ; ディスクアクセスのリトライ
                                     ; 回数を 10 にセット
            jc       error           ; エラー処理
```

INT 21H

一般IOCTL（ハンドル用）

コール

AH　　=44H
AL　　=0CH
BX　　=ハンドル
CH　　=05H　カテゴリコード（プリンタデバイス）
CL　　=ファンクション（マイナー）コード
DS：DX=データバッファへのポインタ

リターン

キャリーフラグがセットされた場合
AX = 01H　無効なファンクションコード

キャリーフラグがセットされない場合
エラーなし

解　説

このシステムコールは、〝PRINT TIL BUSY〞がサポートされているプリンタドライバに対して、プリンタへの出力の繰り返し回数を、設定または取得します。

CL = 45H であると、このコールは、プリンタに対する繰り返し回数をセットします。CL = 65H であると、このコールは、プリンタに対する繰り返し回数を取得します。

DS：DX は、〝PRINT TIL BUSY〞ループの繰り返し回数が格納されているワードを指します。これは、デバイスドライバがデバイスから、〝READY〞シグナルが返されるまでデバイス BUSY を待つ回数です。

INT 21H

一般 IOCTL（ブロックデバイス用）

コール

AH　　＝44H
AL　　＝0DH
BL　　＝デバイス番号（00H　＝カレント、01H　＝A：、…、など）
CH　　＝08H　カテゴリ（メジャー）コード
CL　　＝ファンクション（マイナー）コード
DS：DX＝パラメータブロック–1へのポインタ

リターン

キャリーフラグがセットされた場合
AX ＝01H　無効なファンクションコード
　 ＝05H　アクセスの否定
　 ＝0FH　無効なドライブ

キャリーフラグがセットされない場合
エラーなし

解　　説

IOCTL（ブロックデバイス）に対し、以下のような処理を行います。
処理の種類は、CLでファンクションコードを指定することによって行います。

コード	解　　説
40H	デバイスパラメータのセット
60H	デバイスパラメータの取得
41H	論理デバイス上のトラックのライト（書き込み）
61H	論理デバイス上のトラックのリード（読み出し）
42H	論理デバイス上のトラックのフォーマット
62H	論理デバイス上のトラックのベリファイ

注意　論理デバイスのリード、ライト、フォーマット、ベリファイの前に、デバイスパラメータのセットをしなければなりません。
論理デバイスのリード、ライト、フォーマットまたはベリファイを行いたいときは、次の手順で行ってください。

1. デバイスパラメータの取得を使用して、ドライブパラメータをセーブする。
2. デバイスパラメータのセットを使用し、希望するドライブパラメータを設定する。
3. I/O オペレーションを実行する。
4. デバイスパラメータのセットを使用し、オリジナルのドライブパラメータを復元する。

デバイスパラメータのセット（CL ＝ 40H）

CL = 40H のとき、パラメータブロックは、次のようなフィールドフォーマットになっています。

サイズ	格納データ	
バイト	特殊ファンクション	
バイト	デバイスタイプ	パラメータブロック
ワード	デバイス属性	
ワード	シリンダ数	
バイト	メディアタイプ	
	デバイス BPB	
	トラックレイアウト	

これらのフィールドは、次のような意味をもちます。

・特殊ファンクションフィールド（パラメータブロック－ 1）

各ビットごとの値と意味は次のとおりです。

ビット	値	意　味
0	0	デバイス BPB フィールドには、このデバイスに対する新しいデフォルトの BPB を含んでいる。もし、デバイスのセットのコールが、以前にこのビットをセットしていると、BUILD BPB は実際のメディア BPB を返し、さもなければ、デバイスに対するデフォルト BPB を返す。
	1	すべての BUILD BPB リクエストの結果として、デバイス BPB が返される。
1	0	パラメータブロックのすべてのフィールドのリード。
	1	トラックレイアウトフィールドを除く、すべてのフィールドのパラメータが無視される。
2	0	トラック上のセクタサイズが同じでない（この設定は使用すべきではない）。
	1	トラック上のセクタサイズはすべて同じであり、セクタ番号の範囲は、1から現在のトラック上の総数までである。このビットは、常に設定するべきである。
3～7	0	これらのビットは、0 でなければならない。

・デバイスタイプフィールド

このバイトは、物理デバイスを記述し、デバイスによってセットされます。その値と意味は次のとおりです。

値	意　味
0	320/360Kバイト
1	—
2	640K/720Kバイト
3	256Kバイト（8インチ単密度）
4	1メガバイト
5	固定ディスク、または光ディスク
6	—
7	その他

・デバイス属性フィールド

各ビットごとの値と意味は次のとおりです。

ビット	値	意　味
0	0	メディアは、交換可能。
	1	メディアは、交換不可能。
1	0	ディスクチェンジラインは、サポートされていない（ドアロックがサポートされていない）。
	1	ディスクチェンジラインは、サポートされている（ドアロックがサポートされている）。
2〜15	0	これらのビットは0でなければならない。

・シリンダ数フィールド

このフィールドは、物理デバイスがサポートできるシリンダ数の最大値を示します。この情報は、デバイスによってセットされます。

・メディアタイプフィールド

複数の種類のメディアが使用可能なドライブのために、このフィールドは（デバイスに依存）、どの種類のメディアがドライブにセットされているかを示します。

・デバイスBPBフィールド

特殊ファンクションフィールドのビット0がクリアされた場合、このフィールドのBPBはデバイスの新しいデフォルトのBPBです。

特殊ファンクションフィールドのビット0がセットされた場合、デバイスドライバは、BUILD BPBリクエストの後で、このフィールドにBPBを返します。

・トラックレイアウトフィールド

このフィールドは、各論理デバイスの可変長テーブルと、期待されるメディアトラック上のセクタの

レイアウトを示します。このフィールドのフォーマットは次のとおりです。

データ	種類	内　容
ワード	セクタカウント	セクタの総数
ワード	セクタ番号	セクタ1
ワード	セクタサイズ	セクタ1
ワード	セクタ番号	セクタ2
ワード	セクタサイズ	セクタ2
・	・	・
ワード	セクタ番号	セクタn
ワード	セクタサイズ	セクタn

セクタカウントフィールドは、セクタの総数を示します。各セクタ番号は、1から数えるセクタ総数(n) でなければなりません。

特殊ファンクションフィールドのビット2がセットされているとき、すべてのセクタサイズが同じでなければなりません。

デバイスパラメータの取得（CL　＝60H）

CL＝60Hのとき、パラメータブロックフィールドはCL＝40Hのように、同じフィールドレイアウトです。しかし、いくつかのフィールドは、次のような異なった意味をもっています。

・特殊ファンクションフィールド（パラメータブロック－1）

各ビットごとの値と意味は次のとおりです。

ビット	値	意　味
0	0	デバイスに対するデフォルトのBPBを返す。
	1	BUILD BPBワードが返したBPBを返す。
1〜7	0	これらのビットは0でなければならない。

・トラックレイアウトフィールド

デバイスパラメータの取得コールは、このフィールドを使用しません。

論理デバイス上のトラックのリード／ライト（CL＝61H/CL＝41H）

論理デバイス上のトラックへの書き込みは、CL＝41Hをセットします。論理デバイス上のトラックを読み出すには、CL＝61Hをセットします。

CL＝41HまたはCL＝61Hのとき、パラメータブロックのフォーマットは次のとおりです。

サイズ	内　　容	
バイト	特殊ファンクション	
ワード	ヘッド	パラメータブロック
ワード	シリンダ	
ワード	第1セクタ	
ワード	セクタ数	
2ワード	転送アドレス	

これらのフィールドの内容は次のとおりです。

・特殊ファンクションフィールド（パラメータブロック－1）
このバイトは0です。

・ヘッドフィールド
このフィールドは、書き込み、または読み出しを行うときのヘッド番号。

・シリンダフィールド
このフィールドは、書き込み、または読み出しを行うときのシリンダ番号。

・ファーストセクタフィールド
このフィールドには、書き込み、または読み出しを行うときの最初のセクタ番号があります。このセクタ番号は0から数えるため、4番目のセクタは3になります。

・セクタ番号フィールド
このフィールドは、セクタの総数。

・転送アドレスフィールド
このフィールドは、格納されている書き出すべきデータ、または現在読み出されているデータのアドレス。

論理デバイス上のトラックのフォーマット／ベリファイ（CL　＝42H/CL　＝62H）

論理デバイス上のトラックを、フォーマットとベリファイする場合、CL＝42Hをセットします。論理デバイス上のトラックをベリファイする場合は、CL＝62Hをセットします。

CL＝42HまたはCL＝62Hのとき、パラメータブロックのフォーマットは次のとおりです。

サイズ	内　　容	
バイト	特殊ファンクション	
ワード	ヘッド	パラメータブロック
ワード	シリンダ	

これらのフィールドの意味は次のとおりです。

・特殊ファンクションフィールド（パラメータブロック－1）
このバイトは、0でなければなりません。

・ヘッドフィールド
このフィールドは、フォーマットまたはベリファイを実行するヘッド番号。

・シリンダフィールド
このフィールドは、フォーマットまたはベリファイを実行するシリンダ番号。

INT 21H

ファンクション
440E,0FH 論理ドライブマップの取得／設定

コール

AH ＝ 44H
AL ＝ 0EH　論理ドライブマップの取得
　＝ 0FH　論理ドライブマップの設定
BX ＝ドライブ番号（00H＝カレント、01H ＝ A：、…、など）

リターン

キャリーフラグがセットされた場合
　AX ＝ 01H　無効なファンクションコード
　　＝ 0FH　無効なドライブ

キャリーフラグがセットされない場合
　AL ＝論理デバイスは物理的にマップされた（＝ 0、1ドライブがこの物理ドライブに割り当てられた）

解　説

論理ドライブの取得は、物理ドライブがどの論理ドライブにマップされているかを、MS–DOS に問い合わせます。

論理ドライブマップの設定は、現在物理デバイスにマップされているドライブを変更して行います。これらのファンクションは、ディスクドライブが1台のシステムでのみ有効です。

アプリケーションでは、これらのファンクションを使って、DOS が現在認識しているドライブ中の正しいフロッピィディスクの場所を無効にして、他の論理ドライブをアクセスすることができます。

論理ドライブが、現在どの物理デバイスにマップされているかは、ファンクション 440EH または 440FH（論理ドライブマップの取得／設定）のコールの後で AL の値を調べます。

INT 21H

ファンクション 45H ファイルハンドルの二重化

コール

AH ＝45H
BX ＝ファイルハンドル

リターン

キャリーフラグがセットされた場合
AX ＝04H　オープンされているファイルが多すぎる
　　＝06H　無効なハンドル

キャリーフラグがセットされない場合
AX ＝新規のファイルハンドル

解　説

1つのファイルに追加するハンドルを作成します。BXは、オープンされたファイルのハンドルです。

MS-DOSは新しいハンドルをAXに返します。BXで指定した、すでにオープンされているファイルハンドルを取り出し、同じファイルを示す新規のファイルハンドルを返します（2つのファイルのリード／ライトポインタは同じところを指します）。

このファンクションの実行後、どちらか一方のリード／ライトポインタを移動すると、もう一方のポインタも移動します。このファンクションは、通常、標準入力（ハンドル0）と標準出力（ハンドル1）を、リダイレクトとして扱います。

エラーが起こるとキャリーフラグがセットされ、AXにエラーコードが返されます。

マクロ定義

```
xdup        macro   handle
            mov     bx, handle
            mov     ah, 45H
            int     21H
            endm
```

サンプル

次のプログラムは、標準出力（ハンドル1）を〝DIRFILE〟というファイルに定義しなおし、ディレクトリを出力するための子プロセスを起動して、標準入力をハンドル１に戻します。

```
pgm_file    db      "command.com", 0
cmd_line    db      9, "/c dir/w", 0DH
parm_blk    db      14 dup(0)
path        db      "dirfile", 0
dir_file    dw      ?                ；ハンドル用
sav_stdout  dw      ?                ；ハンドル用
;
func_45H    set_block  last_inst     ；割り当てられたブロックの変更（4AH）
            jc      error_setblk
            create_handle path, 0    ；ハンドルを使うファイルの作成（3CH）
            jc      error_create
            mov     dir_file, ax     ；ハンドルをセーブ
            xdup    1                ；ファイルハンドルを二重化
            jc      error_xdup
            mov     sav_stdout, ax   ；ハンドルをセーブ
            xdup2   dir_file, 1      ；ハンドルを強制的に二重化（46H）
            jc       error_xdup2
            exec pgm_file, cmd_line, parm_blk  ；子プロセスを起動（4BH）
            jc       error_exec
            xdup2    sav_stdout, 1   ；ハンドルを強制的に二重化（46H）
            jc       error_xdup2
            close_handle sav_stdout  ；ハンドルを使うファイルのクローズ（3EH）
            jc       error_close
            close_handle dir_file    ；ハンドルを使うファイルのクローズ（3EH）
            jc       error_close
```

INT 21H

ファンクション 46H ファイルハンドルの強制二重化

コール

AH ＝46H
BX ＝既存のファイルハンドル
CX ＝新規のファイルハンドル

リターン

キャリーフラグがセットされた場合
AX ＝04H　オープンされているファイルが多すぎる
　 ＝06H　無効なハンドル

キャリーフラグがセットされない場合
エラーなし

解説

オープンしたファイルと、すでに連結（二重化）されている他のハンドルを、指定されたハンドルと強制的に二重化させます。BX にはオープンされたファイルのハンドル、CX には新規のハンドルを指定します。

すでにオープンされているファイルハンドルを取り出し、同じ位置の同じファイルを示す新規のファイルハンドルを返します。CX のファイルハンドルが、すでにオープンされていると、そのハンドルがクローズされます。

このファンクションの実行後、どちらか一方のリード／ライトポインタを移動すると、もう一方のポインタも移動します。このファンクションは、通常、標準入力（ハンドル 0）と標準出力（ハンドル 1）を、リダイレクトとして扱います。

エラーが起こるとキャリーフラグがセットされ、AX にエラーコードが返されます。

マクロ定義

```
xdup2       macro   handle1, handle 2
            mov     bx, handle1
            mov     cx, handle2
            mov     ah, 46H
            int     21H
            endm
```

サンプル

次のプログラムは、標準出力（ハンドル1）を〝DIRFILE〟というファイルに定義しなおし、ディレクトリを出力するための子プロセスを起動して、標準入力をハンドル1に戻します。

```
pgm_file     db      "command.com", 0
cmd_line     db      9, "/c dir/w", 0DH
parm_blk     db      14 dup(0)
path         db      "dirfile", 0
dir_file     dw      ?              ; ハンドル用
sav_stdout   dw      ?              ; ハンドル用
;
func_46H: set_block last_inst   ; 割り当てられたメモリブロックの変更 (4AH)
          jc      error_setblk
          create_handle path, 0 ; ハンドルを使うファイルの作成 (3CH)
          jc      error_create
          mov     dir_file, ax  ; ハンドルをセーブ
          xdup    1             ; ファイルハンドルを二重化 (45H)
          jc      error_xdup
          mov     sav_stdout, ax  ; ハンドルをセーブ
          xdup2   dir_file, 1     ; ハンドルを強制的に二重化
          jc      error_xdup2
          exec pgm_file, cmd_line, parm_blk  ; 子プロセスを起動 (48H)
          jc      error_exec
          xdup2   sav_stdout, 1 ; ハンドルを強制的に二重化
          jc      error_xdup2
          close_handle sav_stdout ; ハンドルを使うファイルのクローズ (3EH)
            jc      error_close
            close_handle dir_file ; ハンドルを使うファイルのクローズ (3EH)
            jc      error_close
```

INT 21H

ファンクション 47H カレントディレクトリの取得

コール

AH ＝47H
DS：SI ＝64バイトのメモリ領域に対するポインタ
DL ＝ドライブ番号

リターン

キャリーフラグがセットされた場合
AX ＝0FH　無効なドライブ

キャリーフラグがセットされない場合
エラーなし

解　　説

指定したドライブのカレントディレクトリのパス名を返します。DLは、ドライブ番号（00H＝デフォルト、01H＝A：、…）でなければなりません。SIは、64バイトのメモリ領域のオフセットアドレス（DSは、セグメントアドレス）です。

DS：SIで指定するメモリ領域は、ルートディレクトリからの相対位置で表すパス名（DLで指定したドライブのカレントディレクトリ）の文字列を、ASCIIZ文字列にしたものです。この文字列は、¥マーク（ルートディレクトリを表す）から始まらず、ドライブの指定も含んでいません。

エラーが起こるとキャリーフラグがセットされ、AXにエラーコードが返されます。

マクロ定義

```
get_dir     macro   drive, buffer
            mov     dl, drive
            mov     si, offset buffer
            mov     ah, 47H
            int     21H
            endm
```

サンプル

次のプログラムは、ドライブB上のディスクのカレントディレクトリを表示します。

```
disk        db      "b:$"
buffer      db      64 dup(?)
```

```
;
func_47H:    get_dir 2, buffer          ；カレントディレクトリを得る
             jc      error_dir
             display disk               ;disk を画面に表示 (09H)
             display_asciiz buffer       ；章末参照
```

INT 21H

ファンクション
48H メモリの割り当て

コール

AH = 48H
BX =割り当てるメモリの大きさ（パラグラフ）

リターン

キャリーフラグがセットされた場合
AX = 07H　プログラムの不正なメモリアクセスによって、メモリ中のデータが破壊されている
　　= 08H　メモリが足りない
BX =割り当て可能な最大のメモリサイズ

キャリーフラグがセットされない場合
AX =割り当てられたメモリのセグメントアドレス（パラグラフ）

解説

カレントプロセスに、指定された大きさのメモリを割り当てます。BX には割り当てるメモリの大きさ（パラグラフ単位：1 パラグラフ＝ 16 バイト）を設定します。

要求を満たすメモリがあると、AX に割り当てられたメモリのセグメントアドレスを返します。要求されたメモリがないと、BX に割り当て可能な最大のメモリサイズ（パラグラフ単位）を返します。

エラーが起こるとキャリーフラグがセットされ、AX にエラーコードが返されます。

マクロ定義

```
allocate_memory     macro   bytes
                    mov     bx, bytes
                    mov     cl, 4
                    shr     bx, cl
                    inc     bx
                    mov     ah, 48H
                    int     21H
                    endm
```

サンプル

次のプログラムは、〝TEXTFILE.ASC〞というファイルをオープンし、ファンクション 42H（ファイルポインタの移動）によってサイズを求めます。次に、そのファイルサイズでメモリブロックを割り当て、割り当てたメモリにファイルを読み出します。最後に、割り当てたメモリを開放します。

```
path        db     "textfile.asc", 0
msg1        db     "File loaded into allocated memory block.",
                   0DH, 0AH
msg2        db     "Allocated memory now being freed(deallocated).",
                   0DH, 0AH
handle      dw     ?
mem_seg     dw     ?
file_len    dw     ?
;
func_48H: open_handle path, 0   ; ハンドルを使うファイルのオープン (3DH)
          jc      error_open
          mov     handle, ax            ; ハンドルをセーブ
          move_ptr  handle, 0, 0, 2     ; ファイルポインタを移動 (42H)
          jc      error_move
          mov     file_len, ax          ; ファイルサイズをセーブ
          set_block  last_inst          ; 割り当てられたメモリブロック
                                        ; の変更 (4AH)
          jc      error_setblk
          allocate_memory file_len      ; メモリを割り当てる
          jc      error_alloc
          mov     mem_seg, ax           ; 新規のメモリのアドレスをセーブ
          move_ptr  handle, 0, 0, 0     ; ファイルポインタを移動 (42H)
          jc      error_move
          push    ds                    ;DS をセーブ
          mov     ax, mem_seg           ; 新規のメモリのセグメント
                                        ; アドレスを得る
          mov     ds, ax                ; 新規メモリに DS をセット
          read_handle cs:handle, 0, cs:file_len
                                        ; 新規に割り当てられた
                                        ; メモリに
                                        ; ファイルを読み込む
          pop     ds                    ;DS をリストア
          jc      error_read
          ;(CODE TO PROCESS FILE GOES HERE)
          write_handle stdout, msg1, 42
                                        ;msg1 を表示 (40H)
          jc      write_error
          free_memory mem_seg     ; 割り当てられたメモリを開放 (49H)
          jc      error_freemem
          write_handle stdout, msg2, 49
          jc      write_error     ;msg2 を表示 (40H)
```

INT 21H

ファンクション
49H 割り当てられたメモリの開放

コール

AH = 49H
ES =開放すべきメモリ領域のセグメントアドレス

リターン

キャリーフラグがセットされた場合
AX = 07H プログラムによるメモリ中のデータの破壊
= 09H 不正なメモリブロックの使用

キャリーフラグがセットされない場合
エラーなし

解説

先にファンクション48H（メモリの割り当て）で割り当てられたメモリブロックを開放（他のプログラムが利用可能な状態）します。ESには、開放されるメモリブロックのセグメントアドレスを設定します。

マクロ定義

```
free_memory macro   seg_addr
            mov     ax, seg_addr
            mov     es, ax
            mov     ah, 49H
            int     21H
            endm
```

サンプル

次のプログラムは、〝TEXTFILE.ASC〞というファイルをオープンし、ファンクション42H（ファイルポインタの移動）によってサイズを求めます。次に、そのファイルサイズでメモリブロックを割り当て、割り当てたメモリにファイルを読み込みます。最後に、割り当てたメモリを開放します。

```
path    db   "textfile.asc", 0
msg1    db   "File loaded into allocated memory block.",
              0DH, 0AH
msg2    db   "Allocated memory now being freed(deallocated).",
```

```
                  0DH, 0AH
handle     dw    ?
mem_seg    dw    ?
file_len   dw    ?
;
func_48H:  open_handle path, 0   ; ハンドルを使うファイルのオープン (3DH)
           jc       error_open
           mov      handle, ax         ; ハンドルをセーブ
           move_ptr  handle, 0, 0, 2 ; ファイルポインタを移動 (42H)
           jc       error_move
           mov      file_len, ax       ; ファイルサイズをセーブ
           set_block  last_inst        ; 割り当てられたメモリブロックの変更
           jc       error_setblk
           allocate_memory file_len   ; メモリの割り当て (48H)
           jc       error_alloc
           mov      mem_seg, ax        ; 新規のメモリのアドレスをセーブ
           move_ptr  handle, 0, 0, 0 ; ファイルポインタを移動 (42H)
           jc       error_move
           push     ds                 ;DS をセーブ
           mov      ax, mem_seg  ; 新規のメモリのセグメントアドレスを得る
           mov      ds, ax             ; 新規メモリを DS でポイントする
           read_handle cs:handle, 0, cs:file_len
                                       ; 新規に割り当てられた
                                       ; メモリにファイルを読み込む
           pop      ds                 ;DS をリストア
           jc       error_read
           ;(CODE TO PROCESS FILE GOES HERE)
           write_handle stdout, msg1, 42   ;msg1 を表示 (40H)
           jc       write_error
           free_memory mem_seg         ; 割り当てられたメモリを開放
           jc       error_freemem
           write_handle stdout, msg2, 49   ;msg2 を表示 (40H)
           jc       write_error
```

INT 21H

ファンクション 4AH 割り当てられたメモリブロックの変更

コール

AH ＝4AH
ES ＝メモリ領域のセグメントアドレス
BX ＝変更したいメモリの大きさ（パラグラフ）

リターン

キャリーフラグがセットされた場合
AX ＝07H　プログラムによるメモリ中のデータの破壊
＝08H　十分な空きメモリがない
＝09H　不正なメモリブロックの使用
BX ＝使用可能な最大の大きさ

キャリーフラグがセットされない場合
エラーなし

解　説

割り当てられたメモリブロックの大きさを変更します。ESには、パラグラフ（1パラグラフ＝16バイト）単位のメモリブロックのセグメントアドレスを設定します。

このファンクションがメモリの拡大に失敗すると、BXは、使用可能な最大のブロック（パラグラフ単位）を返します。

MS–DOSは、利用可能なメモリのすべてをCOM形式のファイルに割り当てるため、このコールは、しばしば割り当てられたプログラムのメモリブロックの初期値の縮小に使われます。

マクロ定義

このマクロは、COM形式のプログラムに割り当てられたメモリブロックの初期値を縮小（整理）します。プログラムの最後の命令に続く最初のバイトのオフセットをパラメータ（last_instはサンプルプログラムを参照してください）として渡し、そのパラメータをパラグラフ単位に換算します。次に、その計算結果に17（1はラウンドアップ用、16は256バイトのスタック用）を加え、SPとBPを、そのスタックのポインタにセットします。

```
set_block   macro   last_byte
            mov     bx, offset last_byte
            mov     cl, 4
            shr     bx, cl
            add     bx, 17
            mov     ah, 4AH
            int     21H
            mov     ax, bx
            shl     ax, cl
            dec     ax
            dec     ax
            mov     sp, ax
            endm
```

サンプル　次のプログラムは、子プロセスを起動し、DIR コマンドを実行します。

```
pgm_file    db      "command.com", 0
cmd_line    db      9, "/c dir/w", 0DH
parm_blk    db      14 dup(?)
reg_save    db      10 dup(?)
;
func_4AH:   set_block   last_inst    ；割り当てられたメモリブロックの変更
            exec        pgm_file, cmd_line, parm_blk, 0
                              ；子プロセスを起動し、DIR コマンドを実行（4BH）
```

INT 21H

プログラムのロードと実行

コール

AH　　= 4BH
AL　　= 00H
DS：DX＝パス名の位置
ES：BX＝パラメータブロックの位置

リターン

キャリーフラグがセットされた場合

AX	= 01H	AL に渡されたファンクションが無効である
	= 02H	指定したファイルが無効
	= 03H	指定したパスが無効
	= 05H	アクセスの否定
	= 08H	メモリが足りない
	= 0AH	環境が 32K バイトを超えている
	= 0BH	指定したファイルのフォーマットが不正である

キャリーフラグがセットされない場合
エラーなし

解　　説

指定したプログラムをメモリにロードし、実行します。

DX で、実行可能なプログラムのドライブ名とパス名を表す、ASCIIZ 文字列のオフセットアドレス（セグメントアドレスは DS）を設定します。BX にはロードのためのパラメータブロックのオフセットアドレス（セグメントアドレスは ES）、AL には 00H を設定します。

このファンクションを実行するには、MS–DOS がプログラムをロードするために十分な空きメモリ領域がなければなりません。すべての空きメモリ領域は、ロードされたときにプログラムに割り当てられるので、ファンクション 4BH, コード 00H を使って、他のプログラムをロードして実行する前に、ユーザーはファンクション 4AH（割り当てられたメモリブロックの変更）を使って、メモリを開放しなければなりません。メモリが他の目的で使用されない限り、このファンクションが実行される前に、カレントプロセスによって縮小しなければなりません。

MS–DOS は、プログラムをロードするために PSP を作成し、ファンクション 4BH が呼ばれた直後に、終了アドレス、<CTRL–C>の抜け出しアドレスをセットします。

次に、パラメータブロックのアドレスの内容を示します。

オフセット	バイト長	意　味
00H	2	渡される環境のセグメントアドレス。00H のとき、親環境のコピーであることを示す。
02H	4	PSP のオフセット 80H のコマンドラインのセグメントアドレス（先の 2 バイト）とオフセットアドレス（続く 2 バイト）。これは、128 バイトを超えない正しいコマンドラインでなければならない。
06H	4	新しい PSP（PSP の詳細については第 4 章を参照）のオフセット 5CH にある FCB のセグメントアドレス（先の 2 バイト）とオフセットアドレス（続く 2 バイト）。
0AH	4	PSP のオフセット 6CH の FCB のセグメントアドレス（先の 2 バイト）とオフセットアドレス（続く 2 バイト）。

プロセス中のオープンされたすべてのファイルは、新しくロードされたプログラムでも使用できます。標準入力、標準出力、外部装置、プリンタの各デバイスの細部にわたる情報も、親プログラムから引き継がれます。

実行環境（たとえば、VERIFY＝ON のような環境変数を表す ASCIIZ 文字列）も親プロセスから渡されます。環境はパラグラフ（16 の倍数）の境界から始まり、1 バイトの 0（ASCIIZ 文字列の終わりも含めて 2 バイトの 00H）で終わる 32K バイト未満の ASCIIZ 文字列です。環境変数の後には、引数の数 1 ワード（バージョン 3.3 では 0001H）とプログラムファイル名を表す ASCIIZ 文字列が続きます。

カレントディレクトリ中にファイルが見つかると、ASCIIZ 文字列には、ファンクション 4BH から渡される実行可能なプログラムのドライブ名とパス名が含まれます。ファイルが設定されたパス中で見つかると、ファイル名はパス情報（プログラムをロードするときにこのエリアを使用します）を加えられたものになります。実行環境アドレスが 0 であると、子プロセスは親プロセスの環境を変化させずに引き継ぎます。

環境のセグメントアドレスは、新しい PSP のオフセット 2CH に置きます。ロードしたプログラムのために、パラグラフの境界を設定し、パラメータブロックの最初の 2 バイトに、環境のセグメントアドレスを置きます。親の環境を受け継いだ場合、パラメータブロックの最初の 2 バイトは、ともに 0 になります。

COMMAND.COM による子プロセスの起動

COMMAND.COM は、次の項目を詳細に管理しています。

パス名の設定
プログラムファイルをコマンドパスを通じて検索する
EXE 形式のプログラムを再配置する

他のプログラムをロードし、実行する方法として、COMMAND.COM による子プロセスのロードと実行（起動）があります。次に、その方法を示します。

/C スイッチを含むコマンドラインを子プロセスに渡し（/C に続くコマンドラインで子プロセスになるプログラムについて知らせます）、COM 形式、または EXE 形式のプログラムを起動します。

/C スイッチをともなうコマンドラインのフォーマットは次のとおりです。

〈長さ〉/C〈コマンド〉〈0DH〉

〈長さ〉は、最後のキャリッジリターン（0DH）を含まないコマンドラインの長さです。
〈コマンド〉は、有効な MS–DOS のコマンド、〈0DH〉は、キャリッジリターンコードです。
アプリケーションが直接他のプログラムを実行するとき（COMMAND.COM の代わりに、ファンクション 4BH を使う他のプログラムを指定した場合）、COMMAND.COM が行うすべての作業をアプリケーションで行わなければなりません。

マクロ定義

```
exec        macro   path, command, parms
            mov     dx, offset path
            mov     bx, offset parms
            mov     word ptr parms[02H], offset command
            mov     word ptr parms[04H], cs
            mov     word ptr parms[06H], 5CH
            mov     word ptr parms[08H], es
            mov     word ptr parms[0AH], 6CH
            mov     word ptr parms[0CH], es
            mov     al, 00H
            mov     ah, 4BH
            int     21H
            endm
```

サンプル

次のプログラムは COMMAND.COM をロードし、/W スイッチを使用して DIR コマンドを実行します。

```
pgm_file     db      "command.com", 0
cmd_line     db      9, "/c dir/w", 0DH
parm_blk     db      14 dup(?)
reg_save     db      10 dup(?)
;
func_4B00H   set_block    last_inst    ; 割り当てられたブロックの変更 (4AH)
             exec         pgm_file, cmd_line, parm_blk, 0
                                       ; プログラムをロードし実行
```

INT 21H

オーバーレイのロード

コール

AH　＝4BH
AL　＝03H
DS：DX＝パス名の位置
ES：BX＝パラメータブロックの位置

リターン

キャリーフラグがセットされた場合
AX ＝01H　無効なファンクション
　　＝02H　指定したファイルが無効
　　＝03H　指定したパスが無効
　　＝05H　アクセスの否定
　　＝0AH　環境が 32K バイトを超えている

キャリーフラグがセットされない場合
エラーなし

解　説

DX には指定されたプログラムファイルのドライブ名と、パス名を表す ASCIIZ 文字列のオフセットアドレス（セグメントアドレスは DS）、BX にはパラメータブロックのオフセットアドレス（セグメントアドレスは ES）、AL には 03H を設定します。

MS-DOS はロードするプログラムが、そのプログラム内にロードする領域をもっているとみなすため、とくにメモリを開放（ファンクション 4AH を使って）する必要はありません。また、PSP は作成されません。

次に、パラメータブロックのアドレスの内容を示します。

オフセット	バイト長	意　味
00H	2	プログラムがロードされるセグメントアドレス
02H	2	リロケーション要素。通常、これはパラメータブロックの最初のワード（2 バイト）と同じ（EXE 形式のプログラムとリロケーションの詳細については付録 A「EXE ファイルの構造とローディング」を参照）。

マクロ定義

```
exec_ovl    macro   path, parms, seg_addr
            mov     dx, offset path
            mov     bx, offset parms
            mov     parms, seg_addr
            mov     parms[02H], seg_addr
            mov     al, e
            mov     ah, 4BH
            int     21H
            endm
```

サンプル

次のプログラムは、リダイレクトの標準入力として〝TEXTFILE.ASC〞というファイルをオープンし、オーバーレイとして、〝BIT.COM〞をロードします。次に、〝BIT.COM〞をコールします。〝BIT.COM〞は、標準入力として〝TEXTFILE.ASC〞を読み込みます。

```
stdin       equ     0
;
file        db      "TEXTFILE.ASC", 0
cmd_file    db      "¥bit.com", 0
parm_blk    dw      4 dup(?)
overlay     label   dword
            dw      0
handle      dw      ?
new_mem     dw      ?
;
func_4B03H: set_block   last_inst ;割り当てられたメモリブロックの変更（4AH）
            jc      setblock_error
            allocate_memory 2000  ;メモリの割り当て（48H）
            jc      allocate_error
            mov     new_men, ax   ;メモリのセグメントアドレスをセーブ
            open_handle file, 0   ;ハンドルを使うファイルのオープン
            jc      open_error
            mov     handle, ax    ;ハンドルをセーブ
            xdup2   handle, stdin ;ファイルのハンドルを二重化（45H）
            jc      dup2_error
            close_handle handle   ;ハンドルを使うファイルのクローズ（3EH）
            jc      close_error
            mov     ax, new_men   ;新規メモリのアドレスをセット
            exec_ovl    cmd_file, parm_blk, ax
                                  ;オーバーレイとして
                                  ;プログラムをロード
```

```
jc          exec_error
call        overlay         ; オーバーレイをコール
free_memory new_men         ; 割り当てられたメモリの開放
jc          free_error
```

INT 21H

ファンクション 4CH プロセスの終了

コール　AH = 4CH
AL =リターンコード

リターン　なし

解　説

プロセスを終了させ、MS–DOSに制御を戻します。ALには、ファンクション4DH（子プロセスからリターンコードを取得する）で、親プロセス、またはERRORLEVELを使ったMS–DOSのIFコマンドから返されるリターンコードを設定します。

MS–DOSは、すべてのオープンしているハンドルをクローズし、現在のプロセスを終了させます。次に、制御を起動したプロセスに返します。

このファンクションは、PSPのセグメントアドレスをCSに設定する必要はありません。

マクロ定義

```
end_process macro   return_code
            mov     al, retuen_code
            mov     ah, 4CH
            int     21H
            endm
```

サンプル

次のプログラムはメッセージを表示し、リターンコード8でMS–DOSに制御を戻します。このプログラムのメインルーチンは、サンプルプログラムを参照してください。

```
message     db  "Displayed by FUNC_4CH example", ODH, OAH, "$"
;
func_4CH:   display message       ;message を画面に表示（09H）
            end_process 8         ; リターンコード 8 でプロセスを終了
code        ends
            end     code
```

INT 21H

ファンクション
4DH 子プロセスからリターンコードを取得

コール　AH ＝ 4DH

リターン　AX ＝抜け出しコード

解　　説

ファンクション 31H（キープロセス）、またはファンクション 4CH（プロセスの終了）で、子プロセスを終了するときに指定するリターンコードを１回だけ返します。コードは、AL に返されます。AH はプログラムの終了する状態で、次のような値です。

AH の値	状　態
0	終了
1	<CTRL–C>による終了
2	致命的エラーによる終了
3	常駐したまま終了（ファンクション 31H）

マクロ定義

```
ret_code    macro
            mov     ah, 4DH
            int     21H
            endm
```

サンプル　返されるコードが状況によって変わるため、プログラムは省略します。

INT 21H

ファンクション
4EH 最初に一致するファイル名の検索

コール

AH　　=4EH
DS：DX=パス名の位置
CX　　=属性

リターン

キャリーフラグがセットされた場合
　AX =02H　ファイルがみつからない
　　=03H　パスがみつからない
　　=12H　これ以上ファイルがない

キャリーフラグがセットされない場合
　エラーなし

解　　説

指定したパス名と最初に一致するエントリを検索します。DX には、パス名を表す ASCIIZ 文字列のオフセットアドレス（DS は、セグメントアドレス）を設定します（ワイルドカードを含むことができます)。CX には、ファイルの検索に使われる属性を設定します（属性の詳細は、1.5「ファイルの属性」を参照してください)。

属性フィールドが隠しファイル、システムファイル、ディレクトリエントリ（02H、04H、10H）のいずれかを1つ以上もってる場合、すべての通常のファイルエントリも検索されます。ボリュームラベルを除いたすべてのディレクトリエントリを検索するには、属性バイトに 16H（隠しファイル＋システムファイル＋ディレクトリエントリ）をセットします。

属性とパス名の一致するディレクトリエントリを捜し出した場合、現在のディスク転送アドレス（DTA）で示されるバッファには、次の値が書き込まれます。

オフセット	長さ	説　明
00H	21	予約。ファンクション 4FH（次に一致するファイル名の検索）用。
15H	1	属性の一致
16H	2	ファイルが最初に書き込まれた時刻
18H	2	ファイルが最初に書き込まれた日付
1AH	2	ファイルサイズの下位ワード（2バイト）
1CH	2	ファイルサイズの下位ワード（2バイト）
1EH	13	ファイル名、区切り記号としてのピリオド、拡張子、00H からなる。空白は詰められるので、拡張子があると、ピリオドによって区切られる。

エラーが起こるとキャリーフラグがセットされ、AX にエラーコードが返されます。

マクロ定義

```
find_first_file    macro   path, attrib
                   mov     dx, offset path
                   mov     cx, attrib
                   mov     an, 4EH
                   int     21H
                   endm
```

サンプル

次のプログラムは、メッセージを表示し、ドライブ B のディスクのカレントディレクトリ上に〝REPORT.ASM〟を検索します。

```
yes         db      "FILE EXISTS.", ODH, OAH, "$"
no          db      "FILE DOES NOT EXIST.", ODH, OAH, "$"
path        db      "b:report.asm", 0
buffer      db      43 dup(?)
;
func_4EH:   set     buffer              ; ディスク転送アドレスのセット（1AH)
            find_first_file path, 0     ; 最初に一致するファイル名の検索
            jc      error_findfirst
            cmp     al, 12H             ; これ以上ファイルがないか?
            je      not_there           ; はいのとき、not_there へ
            display yes                 ;yes を画面に表示（09H)
            jmp     return              ; 処理終了
not_there:  display no                  ;no を画面に表示（09H)
```

INT 21H

ファンクション
4FH 次に一致するファイル名の検索

コール　AH ＝ 4FH

リターン
キャリーフラグがセットされた場合
AX ＝ 12H　これ以上ファイルがない

キャリーフラグがセットされない場合
エラーなし

解説

以前に実行されたファンクション 4EH で指定されたファイル名を、続けて検索します。現在のディスク転送アドレス（DTA）にはファンクション 4EH、または先行するファンクション 4FH が返したファイル情報が残っていなければなりません。ファンクション 4EH をコールした後に、ファンクション 1AH によってディスク転送アドレス（DTA）を変更した場合は、このファンクションをコールする前に、ファンクション 4EH をコールしたときのディスク転送アドレス（DTA）に戻さなければなりません。また、ディスク転送アドレス（DTA）の内容は、ファンクション 4EH を参照してください。

エラーが起こるとキャリーフラグがセットされ、AX にエラーコードが返されます。

マクロ定義

```
find_next_file  macro
                mov     ah, 4FH
                int     21H
                endm
```

サンプル

次のプログラムは、ドライブ B 上のカレントディレクトリ中のすべてのファイル数を表示します。

```
message   db      "No files", 0DH, 0AH, "$"
files     dw      ?
path      db      "b:*.*", 0
buffer    db      43 dup(?)
;
func_4FH: set_dta buffer              ; ディスク転送のアドレスのセット (1AH)
```

```
          find_first_file path, 0 ; 最初に一致するファイルなの検索 (4EH)
          jc      error_findfirst
          cmp     al, 12H          ; これ以上ファイルがないか?
          je      all_done         ; はいのとき、all_done へ
          inc     files            ; いいえのとき、ファイルカウンタを
                                   ; インクリメント
search_dir: find_next_file         ; 次に一致するファイル名の検索
          jc      error_findnext
          cmp     al, 12H          ; これ以上エントリがあるか?
          je      done             ; いいえのとき、done へ
          inc     files            ; はいのとき、ファイルカウンタを
                                   ; インクリメント
          jmp     search_dir       ; そして再びチェック
done:     convert files, 10, message  ; 章末参照
all_done: display message          ;message を画面に表示 (09H)
```

INT 21H

ファンクション 54H ベリファイのステータスの取得

コール AH = 54H

リターン AL =現在のベリファイフラグの値（01H =オン、01H =オフ）

解説

MS-DOS のディスクファイルへの書き込み時のベリファイの有無を返します。そのステータスは AL に返され、0 ならばオフ、1 ならばオンです。
ベリファイフラグの設定については、ファンクション 2EH を参照してください。

マクロ定義

```
get_verify  macro
            mov     ah, 54H
            int     21H
            endm
```

サンプル 次のプログラムは、ベリファイのステータスを表示します。

```
message     db      "Verify", "$"
on          db      "on.", 0DH, 0AH, "$"
off         db      "off.", 0DH, 0AH, "$"
;
func_54H:   display message       ;message を画面に表示（09H）
            get_verify            ; ベリファイのステータスを得る
            cmp     al, 0         ; フラグはオフか?
            jg      ver_on        ; いいえのとき、ver_on へ
            display off           ;off を画面に表示（09H）
            jmp     return        ; 処理終了
ver_on:     display on            ;on を画面に表示（09H）
```

INT 21H

ファンクション 56H ディレクトリエントリの変更

コール

AH　　＝56H
DS：DX＝既存のファイルのパス名の位置
ES：DI ＝新規のパス名の位置

リターン

キャリーフラグがセットされた場合

AX ＝02H　ファイルが存在しない
　 ＝03H　パスが存在しない
　 ＝05H　指定したパスがディレクトリであったか、新規パスが既存のファイルであるか、または、新規パスの作成に失敗した
　 ＝11H　既存パスと新規パスのドライブが異なる

キャリーフラグがセットされない場合
エラーなし

解　　説

ディレクトリエントリを変更することによって、ファイル名を変更します。DX は、変更されるエントリのパス名の ASCIIZ 文字列を表すオフセットアドレス（DS は、セグメントアドレス）です。DI は、変更後のエントリのパス名のオフセットアドレス（ES は、セグメントアドレス）です。

ディレクトリが異なっても、他のディレクトリ上のファイルに変更できます。しかし、ディスクドライブが異なる場合は、変更できません。

このファンクションは、隠しファイル、システムファイル、サブディレクトリを変更することはできません。

マクロ定義

```
rename_file macro   old_path, new_path
            mov     dx, offset  old_path
            push    ds
            pop     es
            mov     di, offset new_path
            mov     ah, 56H
            int     21H
            endm
```

サンプル

次のプログラムは、変更前と変更後のファイル名を表示し、ファイル名を変更します。

```
prompt1   db      "Filename: $"
prompt2   db      "New name: $"
old_path  db      15, ?, 15 dup(?)
new_path  db      15, ?, 15 dup(?)
crlf      db      0DH, 0AH, "$"
;
func_56H: display prompt1            ;prompt1 を画面に表示 (09H)
          get_string 15, old_path ; 変更前パス名をキーボード入力 (0AH)
          xor     bx, bx             ;BL はインデックスとして使用
          mov     bl, old_path[1] ; 文字列長を得る
          mov     old_path[bx+2], 0   ;ASCIIZ 文字列を作成
          display crlf               ;crlf を画面に出力 (09H)
          display prompt2            ;prompt2 を画面に表示 (09H)
          get_string 15, new_path     ; 変更後パス名をキーボード入力 (0AH)
          xor     bx, bx             ;BL はインデックスとして使用
          mov     bl, new_path[1] ; 文字列長を得る
          mov     old_path[bx+2], 0   ;ASCIIZ 文字列を作成
          display crlf               ;crlf を画面に出力 (09H)
          rename_file old_path[2], new_path[2]
                                     ; ディレクトリエントリを
                                                  ; 変更
          jc      error_rename                    ; エラー処理
```

INT 21H

ファンクション 57H　ファイルの日付／時刻の取得／設定

コール

AH ＝57H
AL ＝00H　日付／時刻を取得する
　　＝01H　日付／時刻を設定する
BX ＝ファイルハンドル
AL ＝01Hの場合
　CX ＝　　セットすべき時刻
　DX ＝　　セットすべき日付

リターン

キャリーフラグがセットされた場合
　AX ＝01H　　無効なファンクション
　　　＝06H　　オープンされていないハンドルへのアクセス

キャリーフラグがセットされない場合
　AL ＝00H　　CX/DXに最後に編集された日時が返される
　　　＝01H　　エラーなし

解　　説

ファイルが最後に編集された日付と時刻を取得、または設定します。日付と時刻を得る場合、ALを00Hにして呼び出します。結果は、CXとDXに時刻と日付が返されます。日付と時刻を設定する場合は、ALを01Hに、CXとDXには時刻と日付を設定して呼び出します。BXはファイルハンドルです。日付と時刻については、1.8「ファイルコントロールブロック」を参照してください。

マクロ定義

```
get_set_date_time   macro   handle, action, time, date
                    mov     bx, handle
                    mov     al, action
                    mov     cx, word ptr time
                    mov     dx, word ptr date
                    mov     ah, 57H
                    int     21H
                    endm
```

サンプル

次のプログラムは、ドライブBのディスク上の〝REPORT.ASM〞を得て、その日を翌日に更新し（変更される日付が1日を超える場合、年と月も変更されます）、新しい日付をファイルにセットします。

```
month       db     31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
path        db     "b:report.asm", 0
handle      dw     ?
time        db     2 dup(?)
date        db     2 dup(?)
;
func_57H: open_handle path, 0       ; ハンドルを使うファイルのオープン (3DH)
          mov     handle, ax        ; ハンドルのセーブ
          get_set_date_time  handle, 0, time, date
                                      ; ハンドルの日付／時刻を得る
          jc      error_time
          mov     word ptr time, cx   ; 時刻をセーブ
          mov     word ptr date, dx   ; 日付をセーブ
          convert_date  date[-24]     ; 章末参照
          inc     dh                  ; 日をインクリメント
          xor     bx, bx              ;BL はインデックスとして使用
          mov     bl, dl              ; 月を得る
          cmp     dh, month[bx-1]     ; 月の最終日を越えているか?
          jle     month_ok            ; いいえのとき、month_ok へ
          mov     dh, 1               ; はいのとき、日に 1 をセット
          inc     dl                  ; 月をインクリメント
          cmp     dl, 12              ; 月は 12 を越えているか?
          jle     month_ok            ; いいえのとき、month_ok へ
          mov     dl, 1               ; はいのとき、月を 1 セット
          inc     cx                  ; 年をインクリメント
month_ok: pack_date  date             ; 章末参照
          get_set_date_time  handle, 1, time, date
                                      ; ファイルの日付／時刻を得る
          jc      error_time
          close_handle  handle      ; ハンドルを使うファイルのクローズ (3EH)
          jc      error_close
```

INT 21H

ファンクション 58H アロケーションストラテジの取得／設定

コール

AH ＝ 58H
AL ＝ 00H　ストラテジを得る
　 ＝ 01H　ストラテジを設定する

AL ＝ 01H の場合
BX ＝ 00H　下位
　 ＝ 01H　最小
　 ＝ 02H　上位

リターン

キャリーフラグがセットされた場合
AX ＝ 01H　無効なファンクションコード

キャリーフラグがセットされない場合
AL ＝ 00H の場合
AX ＝ 00H　下位
　 ＝ 01H　最小
　 ＝ 02H　上位

解　説

ALが00Hの場合、AXにストラテジを返します。ALが01Hの場合、BXはストラテジでなければなりません。次にストラテジのステータスを示します。

値	名前	意　味
00H	下位	MS–DOSはデフォルトとして、最も下位の利用可能なブロックから捜し始め、最初に見つかったブロックを割り当てる（割り当てられたメモリは、最も下位の利用可能なブロック）。
01H	最小	MS–DOSは、利用可能な各ブロックを捜し、必要最小の利用可能なブロックを割り当てる。
02H	上位	MS–DOSは、最も上位の利用可能なブロックから捜し始め、最初に見つかったブロックを割り当てる（割り当てられたメモリは、最も上位の利用可能なブロック）。

このファンクションリクエストは、MS–DOSのメモリの管理を制御できます。

エラーが起こるとキャリーフラグがセットされ、AXにエラーコードを返します。

マクロ定義

```
alloc_strat macro    code, strategy
            mov      bx, strategy
            mov      al, code
            mov      ah, 58H
            int      21H
            endm
```


サンプル

次のプログラムは、実際のメモリアロケーションストラテジを表示し、ストラテジを上位（2）に設定することによって、次に割り当てられるメモリを、利用可能なメモリの最も上位のものにします。

```
get        equ      0
set        equ      1
stdout     equ      1
last_fit   equ      2
;
first      db       "First fit", ODH, OAH
best       db       "Best fit", ODH, OAH
last       db       "Last fit", ODH, OAH
;
func_58H:  alloc_strat get          ; アロケーションストラテジを得る
           jc       alloc_error
           mov      cl, 4           ; オフセットを算出するために
           shl      ax, cl          ; リターンコードを 16 倍する
           mov      dx, offset first ;first メッセージのオフセットをセット
           add      dx, ax          ; そしてベースアドレスを加算
           mov      bx, stdout      ; 書き込むハンドルを指定
           mov      cs, 16          ;16 バイト書き込む
           mov      ah, 40h         ; ファンクションコードを指定
           int      21H             ; システムコール（40H）
           alloc_strat set, last_fit ; アロケーションストラテジをセット
           jc       alloc_error
```

INT 21H

ファンクション 59H 拡張エラーコードの取得

コール

AH ＝ 59H
BX ＝ 00H

リターン

AX ＝拡張エラーコード
BH ＝エラークラス
BL ＝可能な対処
CH ＝エラーの発生した場所
CL、DX、SI、DI、BP、DS、ES の各レジスタの内容は破壊される

解　説

ユーザーが用意した割り込みタイプ 24H のハンドラで、このファンクションを使うと、致命的なエラーの詳細な情報を得ることができます。コールの BX はエラーのレベルを表します。通常は 00H です。

次に、このファンクションの4つのリターン情報（AX、BH、BL、CH の4つのレジスタに返される）の詳細を示します（AX については、エラーコード一覧を参照してください）。

BH ＝エラークラス

BH には、エラーのクラスに関するコードが返されます。次にその内容を示します。

コード	意　味
01H	メモリ容量や I/O チャネルなどの資源の不足
02H	エラーではないが、終了するべき一時的状況（ファイルの一部分がロックされている）に陥っている
03H	アクセス特権のエラー（例：MS–Networks でアクセス特権のないディレクトリへのアクセスエラーなど）
04H	システムソフトウェアの内部エラー
05H	ハードウェアに起因するエラー
06H	現在のプロセスが原因でないシステムソフトウェアのエラー
07H	アプリケーションプログラムのエラー
08H	ファイルまたは項目がない
09H	ファイルまたは項目が、無効なフォーマットかタイプ。さもなければ、ファイルまたは項目が無効か、適切ではない
0AH	ファイルまたは項目が内部的にロックされている
0BH	ドライブ内のディスク上に問題がある。ディスクの一部分か、記憶媒体自身に問題がある
0CH	その他の原因によるエラー

BL =可能な対処

BLには、エラーに対してプログラムが対応できることを示すコードが返されます。

コード	意　味
01H	再試行、ユーザーに確認を求める
02H	休止後に再試行
03H	ドライブ名やファイル名などのデータの入力の場合、ユーザーに再度の入力を求める
04H	メモリの内容をクリアし、終了する
05H	すぐに終了すること。ファイルのクローズやインデックスのアップデートよりも優先して、すぐにプログラムを終了しなければならないほどシステムの状況が異常
06H	エラーコードを参照
07H	ディスクを取り換え、再試行するなどの動作を、ユーザー側で行わなければならない

CH =エラーが発生した場所

CHには、エラーにともなうメモリの種類などの付加情報のコードが返されます。これらは、とくにハードウェアに起因するエラーです（BH = 5）。

コード	意　味
01H	不明
02H	ディスクドライブのような、ランダムアクセスブロックデバイスに関するエラー
03H	ネットワークに関するエラー
04H	プリンタのような、シリアルアクセスキャラクタデバイスに関するエラー
05H	ランダムアクセスメモリ（RAM）に関するエラー

バージョン3.0以前のシステムコールでエラーが発生したら、このシステムコールを実行します。これによって、拡張エラーコードを得ることができます。プログラムが拡張されたエラーコードを使わなくても、バージョン3.0以前のエラーコードで対応できます。

このシステムコールは、割り込みタイプ24Hで利用でき、ネットワーク関係のエラーコードを返すことができます。

マクロ定義

```
get_error   macro   fcb
            mov     ah, 59H
            mov     bx, 0
            int     21H
            endm
```

サンプル

このファンクションリクエストは、割り込みなどの種々の状況を設定しなければならないため、プログラムは省略します。

59H

INT 21H

ファンクション 5AH 一時ファイルの作成

コール

AH　　=5AH
CX　　=属性
DS：DX=1バイトの00Hと13バイトのメモリが続くパス名の位置

リターン

キャリーフラグがセットされている場合
　AX =03H　パス名がない
　　 =05H　アクセスできない

キャリーフラグがセットされない場合
　AX =ファイルハンドル

解　　説

指定した条件で、一時ファイルを作成します。DXには、パス名、00Hとメモリの13バイト（ファイル名を保持している）からなるASCIIZ文字列のオフセットアドレス（セグメントアドレスは、DS）を設定します。CXには、ファイルに割り当てられた属性を設定します（属性については1.5「ファイルの属性」を参照してください）。

MS-DOSは、特別なファイル名を作成し、そのファイル名にDS：DXが指定するパス名を付け加えます。次に、そのファイルを作成し、通常のファイルと互換性のあるモードでオープンし、AXにファイルハンドルを返します。一時的にファイルを必要とするプログラムは、このファンクションを使って重複したファイル名を使用しないようにします。

このファンクションで作成された一時ファイルは、プロセスが終了しても自動的に消去されません。一時ファイルが必要でなくなった時点で消去してください。

エラーが起こるとキャリーフラグがセットされ、AXにエラーコードを返します。

マクロ定義

```
create_temp     macro   pathname, attrib
                mov     cx, attrib
                mov     dx, offset pathname
                mov     ah, 5AH
                int     21H
                endm
```

サンプル

次のプログラムは、ディレクトリ〝¥WP¥DOCS〞に一時ファイルを作成し、カレントディレクトリの〝TEXTFILE.ASC〞を一時ファイル内にコピーし、両方のファイルをクローズします。

```
stdout      equ     1
;
file        db      "TEXTFILE.ASC", 0
path        db      "¥WP¥DOCS", 0
temp        db      13 dup(0)
open_msg    db      "opened", 0DH, 0AH
crl_msg     db      "created.", 0DH, 0AH
rd_msg      db      "read into buffer.", 0DH, 0AH
wr_msg      db      "Buffer written to"
cl_msg      db      "Files closed.", 0DH, 0AH
crlf        db      0DH, 0AH
handle1     dw      ?
handle2     dw      ?
buffer      db      512 dup(?)
;
func_5AH: open_handle     file, 0 ;ハンドルを使うファイルのオープン（3DH）
          jc      open_error
          mov     handle1, ax         ;ハンドルのセーブ
          write_handle    stdout, file, 12     ;file を表示（40H）
          jc      write_error
          write_handle  stdout, open_msg, 10   ;open_msg を表示（40H）
          jc      write_error
          create_temp   path, 0                ;一時ファイルを作成
          jc      create_error
          mov     handle2, ax                  ;ハンドルをセーブ
          write_handle  stdout, path, 8        ;path を表示（40H）
          jc      write_error
          display_char    "¥"                  ;文字〝¥〞を表示（02H）
          write_handle  stdout, temp, 12       ;temp を表示（40H）
          jc      write_error
          write_handle  stdout, crl_msg, 11    ;crl_msg を表示（40H）
          jc      write_error
          read_handle   handle1, buffer, 512   ;ハンドルで指定された
                                               ;ファイルから
                                               ;読み込む（3FH）
          jc      read_error
          write_handle  stdout, file, 12       ;file を表示（40H）
```

5AH

```
jc        write_error
write_handle  stdout, rd_msg, 20      ;rd_msg を表示（40H）
jc        write_error
write_handle  handle2, buffer, 512   ; ハンドルで指定された
                                      ; ファイルへ
                                      ; 書き込む（40H）
jc        write_error
write_handle  stdout, wr_msg, 18      ;wr_msg を表示（40H）
jc        write_error
write_handle  stdout, temp, 12        ;temp を表示（40H）
jc        write_error
write_handle  stdout, crlf, 2         ;crlf を表示（40H）
jc        write_error
close_handle  handle1        ; ハンドルを使うファイルのクローズ
jc        close_error
close_handle  handle2        ; ハンドルを使うファイルのクローズ
jc        close_error
write_handle  stdout, cl_msg, 15      ;cl_msg を表示（40H）
jc        write_error
```

INT 21H

ファンクション 5BH 新しいファイルの作成

コール

AH ＝5BH
CX ＝属性
DS：DX＝パス名の位置

リターン

キャリーフラグがセットされた場合

AX ＝03H パスが存在しない
＝04H オープンされているファイル数が多すぎる
＝05H アクセスの否定
＝50H ファイルがすでに存在している

キャリーフラグがセットされない場合

AX ＝ファイルハンドル

解説

既存のファイルと重複しないように、ハンドルを使うファイルを新規作成します。

DX にはパス名を表す ASCIIZ 文字列のオフセットアドレス（DS は、セグメントアドレス）を、CX には属性を設定します（属性の詳細は 1.5「ファイルの属性」を参照してください）。

同じファイル名が存在しない限り、MS–DOS はファイルを作成しオープンして、AX にハンドルを返します。

ファンクション 3CH（ハンドルを使うファイルの作成）は、同じファイル名が存在すると、ファイルの内容が 0 バイトのファイル名を作成しますが、このファンクションは、エラーを返します。また、ファイルの存在は、マルチタスクシステムのセマフォとして使えますので、このシステムコールはセマフォのテストとセットに使用できます。

マクロ定義

```
create_new  macro  pathname, attrib
            mov    cx, attrib
            mov    dx, offses pathname
            mov    ah, 5BH
            int    21H
            endm
```

サンプル

次のプログラムは、カレントディレクトリに〝REPORT.ASM〞という名の新しいファイルを作成します。同じ名前のファイルが存在するとエラーメッセージを表示し、MS-DOSに戻ります。同じ名のファイルが存在せず、他のエラーがないと、プログラムはハンドルをセーブしプロセスを続行します。

```
err_msg        db       "FILE ALREADY EXISTS", 0DH, 0AH, "$"
path           db       "REPORT.ASM", 0
handle         dw       ?
;
func_5BH:      create_new  path, 0  ; 新しいファイルを作成
               jnc      exec_process  ; エラーのないとき、プロセスを実行
               cmp      ax, 80      ; ファイルは既に存在するか?
               jne      error
               display err_msg      ;err_msg を画面に表示 (09H)
               jmp      return      ;MS-DOS に戻る
exec_process: mov      handle, ax   ; ハンドルのセーブ
;
;              (further processing here)
```

INT 21H

ファンクション
5C00H ファイルアクセスのロック

コール

AH　＝5CH
AL　＝00H
BX　＝ファイルハンドル
CX：DX＝ロックされた領域のオフセット
SI：DI　＝ロックされた領域の長さ

リターン

キャリーフラグがセットされた場合
AX ＝01H　無効なファンクションコード
＝06H　指定したハンドルは無効か、すでにオープンされている
＝21H　ロックされている領域にアクセスしようとした

キャリーフラグがセットされない場合
エラーなし

解　説

プログラムエリア内の、指定した領域へのアクセスをロックします。

BXにはロックされた領域を含むファイルのハンドル、CX：DX（4バイト整数）にはファイル内のロックされた領域の始めのオフセット、SI：DI（4バイト整数）には領域の長さを設定します。

他のプロセスがロックされた領域をアクセス（読み出しか書き込み）しようとすると、MS-DOSは3回再試行し、失敗すると、そのプロセスのために割り込みタイプ24H（致命的エラーによる中断アドレス）を実行します。再試行の回数の変更は、ファンクション440BHを参照してください。

ロックされた領域は、EOFを超えていてもエラーにはなりません。

ファンクション45H（ファイルハンドルの二重化）と46H（ファイルハンドルの強制二重化）は、ロックされた領域に関してもアクセスします。ファンクション4B00H（プログラムのロードと実行）を使って、子プロセスにオープンファイルを渡しても、ロックされた領域に多重アクセスすることはできません。

プログラムがロックされた領域を含むファイルを閉じるか、またはロックされた領域を含むファイルをオープンしたまま終了した場合、結果は保証されません。割り込みタイプ23H（<CTRL-C>）、24H（致命的エラー）によって終了するプログラムは、割り込みタイプを回避するか、または終了する前にロックされた領域をアンロック（解除）します。

プログラムはロックされた領域がアクセスできないことを確認できません。領域をロックしようとしてエラーコードを確認することによって、プログラムは、領域のステータス（ロックされているか否か）を確認することができます。

マクロ定義

```
lock        macro   handle, start, bytes
            mov     bx, handle
            mov     cx, word ptr start
            mov     dx, word ptr start+2
            mov     si, word ptr bytes
            mov     di, word ptr bytes+2
            mov     al, 0
            mov     ah, 5CH
            int     21H
            endm
```

サンプル

次のプログラムは、ロックされていない〝FINALRPT〞という名のファイルをオープンし、最初の128バイトと1024バイトから5116バイトまでの2箇所をロックします。この後、同じ場所をアンロックし、クローズします。

```
stdout      equ     1
;
start1      db      0
lgth1       db      128
start2      db      1023
lgth2       db      4096
file        db      "FINALRPT", 0
op_msg      db      "opened.", 0DH, 0AH
11_msg      db      "First 128 bytes locked.", 0DH, 0AH
12_msg      db      "Bytes 1024-5119 locked.", 0DH, 0AH
ul_msg      db      "First 128 bytes unlocked.", 0DH, 0AH
u2_msg      db      "Bytes 1024-5119 unlocked.", 0DH, 0AH
cl_msg      db      "closed.", 0DH, 0AH
handle      dw      ?
;
func_5C00H: open_handle file, 01000010b     ;ハンドルを使う
                                            ;ファイルのオープン (3DH)
            jc      open_error
            write_handle  stdout, file, 8   ;file を表示 (40H)
            jc      write_error
            write_handle  stdout, op_msg, 10  ;op_msg を表示 (40H)
            jc      write_error
            mov     handle, ax              ;ハンドルをセーブ
            lock    handle, start1, lgth1   ;ファイルアクセスのロック
            jc      lock_error
            write_handle  stdout, 11_msg, 25  ;11_msg を表示 (40H)
```

```
                jc       write_error
                lock     handle, start2, lgth2 ; ファイルアクセスのロック
                jc       lock_error
                write_handle  stdout, l2_msg, 25  ;l2_msg を表示
                jc       write_error
;
;               (further processing here)
;
                unlock   handle, start1, lgth1      ; ファイルアクセスの
                                                    ; ロックを解除 (5C01H)
                jc       unlock_error
                write_handle  stdout, ul_msg, 27  ;ul_msg を表示 (40H)
                jc       write_error
                unlock   handle, start2, lgth2 ; ファイルアクセスの
                                               ; ロックを解除 (5C01H)
                jc       unlock_error
                write_handle  stdout, u2_msg, 27  ;u2_msg を表示 (40H)
                jc       write_error
                close_handle  handle           ; ハンドルを使うファイルの
                                               ; クローズ (3EH)
                jc       close_error
                write_handle  stdout, file, 8 ;file を表示 (40H)
                jc       write_error
                write_handle  stdout, cl_msg, 10  ;cl_msg を表示
                jc       write_error
```

5C00H

INT 21H

ファイルアクセスのロック解除

コール

AH　　＝5CH
AL　　＝01H
BX　　＝ハンドル
CX：DX＝ロックを解除する領域のオフセット
SI：DI ＝ロックを解除する領域の長さ

リターン

キャリーフラグがセットされた場合
AX ＝01H 無効なコード。またはファイルシェアリング（SHARE.EXE）が常駐していない
＝06H 指定したハンドルが無効か、すでにオープンされている
＝21H 指定した領域は、ファンクション5C00Hでロックされた領域ではない

キャリーフラグがセットされない場合
エラーなし

解　　説

ファンクション5C01Hでロックした領域を開放します。

BXにはロックを解除する領域を含むファイルのハンドル、CX：DX（4バイト整数）にはファイル内のロックされた領域の始めのオフセット、SI：DI（4バイト整数）には領域の長さを設定します。このオフセットと領域の長さは、ファンクション5C00H（ロック）でロックされたときに指定されたものと同じでなければなりません。

ロックされる領域については、ファンクション5C00H（ロック）を参照してください。

マクロ定義

```
unlock      macro   handle, start, bytes
            mov     bx, handle
            mov     cx, word ptr start
            mov     dx, word ptr start+2
            mov     si, word ptr bytes
            mov     di, word ptr bytes+2
            mov     al, 1
```

```
mov     ah, 5CH
int     21H
endm
```

サンプル　　ファンクション 5CH、コード 00H を参照してください。

INT 21H

ファンクション 5E00H　マシン名の取得

コール

AH　　＝5EH
AL　　＝00H
DS：DX＝16バイトのバッファの位置

リターン

キャリーフラグがセットされた場合
　AX ＝01H　　無効なファンクションコード

キャリーフラグがセットされない場合
　CX ＝ローカルコンピュータの番号

解　　説

このファンクションは、ローカルコンピュータのネット名を得ます。DXには、16バイトのバッファのオフセットアドレス（DSは、セグメントアドレス）を設定します。MS-Networksの稼働が必要です。

MS-DOSは、DS：DXの指定するバッファ中のローカルコンピュータ名（16バイトのASCIIZ文字列。ブランクは詰めます）を返します。CXは、ローカルコンピュータの番号を返します。

マクロ定義

```
get_machine_name    macro   buffer
                    mov     dx, offset buffer
                    mov     al, 0
                    mov     ah, 5EH
                    int     21H
                    endm
```

サンプル

次のプログラムはMS-Networksのワークステーションの名前を表示します。

```
stdout       equ      1
;
msg          db       "Netname:"
mac_name     db       16 dup(?), 0DH, 0AH
;
func_5E00H: get_machine_name mac_name        ; ワークステーションの
```

```
                                        ; 名前を得る
        jc          name_error
        write_handle  stdout, msg, 27   ;msg を表示 (40H)
        jc          write_error
```

INT 21H

ファンクション 5E02H　プリンタセットアップ

コール

AH　　＝5EH
AL　　＝02H
BX　　＝割り当てリストのインデックス
CX　　＝セットアップ文字列の長さ
DS：SI ＝セットアップ文字列の位置

リターン

キャリーフラグがセットされた場合
　AX ＝ 01H　無効なファンクションコード。または、MS–Networks が稼働していない

キャリーフラグがセットされない場合
　エラーなし

解　説

ネットワークプリンタに送る各ファイルの先頭に、MS–DOS が付けるコントロールキャラクタを定義します。BX にはプリンタの割り当てリストの中のインデックス（エントリ 0 は、最初のエントリになります)、CX にはセットアップ文字列の長さ、SI にはセットアップ文字列のオフセットアドレス（DS は、セグメントアドレス）を設定します。MS–Networks の稼働が必要です。

セットアップ文字列は、BX の割り当てリストのインデックスによって、プリンタに送られる各ファイルの先頭に付け加えます。このファンクションリクエストは、プリンタコンフィグレーションをもったプリンタを受けもつプログラムで使われます。ファンクション 5F02H を使って、プリンタの割り当てリストを登録することができます。

マクロ定義

```
printer_setup   macro   index, lgth, string
                mov     bx, index,
                mov     cx, lgth
                mov     dx, offset string
                mov     al, 2
                mov     ah, 5EH
                int     21H
                endm
```

サンプル

各プリンタに依存するため、プログラムは省略します。

INT 21H

割り当てリストのエントリの取得

コール

AH　＝5FH
AL　＝02H
BX　＝割り当てリストのインデックス
DX：SI ＝ローカル名のバッファの位置
ES：DI ＝リモート名のバッファの位置

リターン

キャリーフラグがセットされた場合
AX ＝01H　無効なファンクションコード。または、MS–Networksが稼働していない
　＝12H　これ以上ファイルがない

キャリーフラグがセットされない場合
BL ＝03H　プリンタ
　＝04H　ドライブ
CX ＝ユーザー変数域

解　　説

このファンクションは、ネットワークの割り当てリストのエントリを得ます。BXには割り当てリストインデックス（エントリ0のときは最初のエントリ)、SIにはローカル名のための16バイトのバッファのオフセットアドレス（DSは、セグメントアドレス)、DIにはリモート名の128バイトのバッファのオフセットアドレス（ESは、セグメントアドレス）を設定します。MS–Networksの稼動が必要です。

MS–DOSは、DS：SIで指定するバッファ内のローカル名と、ES：DIで指定するバッファ内のリモート名を設定します。ローカル名は、ヌルで終るASCIIZ文字列になります。BLは、ローカルデバイスがプリンタの場合は03H、デバイスの場合は04Hを返します。CXは、ファンクション5F03H（割り当てリストのエントリの作成）で設定されたユーザー変数の値を返します。割り当てリストは、その内容を書き換えることもできます。

このファンクションリクエストを使って、エントリを得るか、またはテーブルを検索して完成したリストのコピーを作ることができます。割り当てリストの終わりを見つけると、ファンクション4EH（最初に一致するファイル名の検索)、4FH（次に一致するファイル名を検索）でディレクトリを検索するときのように、エラーコード12Hをチェックします。

マクロ定義

```
get_list    macro   index, local, remote
            mov     bx, index
            mov     si, offset local
            mov     di, offset remote
            mov     al, 2
            mov     ah, 5FH
            int     21h
            endm
```

サンプル

次のプログラムは、MS–metworks のワークステーションの各エントリのローカル名、リモート名、デバイスタイプ（ドライブかプリンタ）、割り当てリストを表示します。

```
stdout      equ     1
printer     equ     3
;
local_nm    db      16 dup(?), 2 dup(20h)
remote_nm   db      128 dup(?), 2 dup(20h)
header      db      "Local name", 8 dup(20h)
db          "Remote name", 7 dup(20h)
db          "Device Type"
crlf        db      0dh, 0ah, 0dh, 0ah
drive_msg   db      "drive"
print_msg   db      "printer"
index       dw      ?
;
func_5F02H: write_handle  stdout, header, 51  ;header を表示 (40H)
            jc      write_error
            mov     index, 0             ;割り当てリストのインデックスを設定
            ck_list get_list index, local_nm, remote_nm
                                         ;割り当てリストのエントリを得る
           jnc      got_one              ;1 エントリを得る、got_one へ
error:     cmp      ax, 18               ;ラストエントリか?
           je       last_one             ;はいのとき、last_one へ
           jmp      error
got_one:   push     bx                   ;デバイスタイプをセーブ
           write_handle  stdout, local_nm 148 ;local_nm を表示 (40H)
           jc       write_error
           pop      bx                   ;デバイスタイプをリストア
           cmp      bl, printer          ;プリンタデバイスか?
           je       prntr                ;はいのとき、print へ
```

```
            write_handle  stdout, drive_msg, 5 ;drive_msgを表示
            jc       write_error               ; (40H)
            jmp      get_next                  ;get_nextへ
prntr:      write_handle  stdout, print_msg, 7 ;print_msgを表示
            jc       write_error               ; (40H)
get_next:   write_handle  stdout,  crlf, 2     ;crlfを表示 (40H)
            jc       write_error
            inc      index           ; インデックスをインクリメント
            jmp      ck_list                   ; 次のエントリを得る
last_one:   write_handle  stdout, crlf, 4      ;crlfを表示 (40H)
            jc       write_error
;
            jmp      return
```

INT 21H

割り当てリストのエントリの作成

コール

AH　　=5FH
AL　　=03H
BL　　=03H　プリンタ
　　　=04H　ドライブ
CX　　=ユーザー変数域
DS：SI =ソースデバイス名の位置
ES：DI =ディスティネーションデバイス名の位置

リターン

キャリーフラグがセットされた場合

AX = 01H	無効なファンクションコード。MS–Networksが稼働していない。または、書式に誤りがある
= 03H	パスが無効か、存在しない
= 05H	アクセスの否定
= 08H	ネットワークが起こしたエラーによるメモリ不足

キャリーフラグがセットされない場合
　エラーなし

解説

このファンクションは、プリンタまたはディスクドライブ（ソースデバイス）をネットワークディレクトリ（ディスティネーションデバイス）としてリディレクトします。BLにはソースデバイスがプリンタなら03H、ディスクドライブなら04Hを設定します。

SIにはプリンタ名、コロン付きのドライブ名、ヌル文字列（1バイトの00H）のいずれかを表すASCIIZ文字列のオフセットアドレス（DSは、セグメントアドレス）、DIにはネットワークディレクトリ名を表すASCIIZ文字列のオフセットアドレス（ESは、セグメントアドレス）を設定します。MS–Networksの稼動が必要です。

ディスティネーション文字列は、次のような書式です。

〈マシン名〉〈パス名〉〈00H〉〈パスワード〉〈00H〉

〈マシン名〉は、ネットワークのサーバのネット名で、¥¥で始まる文字列です。

〈パス名〉は、ソースデバイスからリダイレクトして渡されるネットワークディレクトリのエイリアス（別名）です。

〈00H〉はヌルコードです。

〈パスワード〉は、ネットワークをアクセスするためのパスワードです。パスワードがない場合、〈パス名〉の後には、2バイトのヌルコードが続かなければなりません。

BL＝03Hの場合、ソース文字列はPRNでなければなりません。プリンタとして登録されたすべての出力はバッファに貯められ、ディスティネーション文字列に登録されたリモートプリンタスプーラに送られます。

BL＝04Hの場合、ソース文字列はコロン付きのドライブ名か、ヌル文字列のいずれかでなければなりません。ソース文字列が無効なドライブ名とコロンの場合、それ以降のすべてのドライブ名は、ディスティネーション文字列に登録されたネットワークディレクトリにリディレクトに渡されたものと見なします。ソース文字列がヌルの場合、MS–DOSは、パスワードが合うネットワークディレクトリとしてアクセスしようとします。

ディスティネーション文字列は、128バイト以下でなければなりません。CXのユーザー変数は、ファンクション5F02H（割り当てリストエントリを得る）で与えられます。

マクロ定義

```
redir       macro   device, value, source, destination
            mov     bl, device
            mov     cx, value
            mov     si, offset source
            mov     es, seg destination
            mov     di, offset destination
            mov     al, 03H
            mov     ah, 5FH
            int     21H
            endm
```

サンプル

次のプログラムは〝HAROLD〟という名のサーバに、ワークステーションから、2つのデバイスとプリンタをリダイレクトして渡します。マシン名、ディレクトリ名、ドライブ文字は、次のようになります。

ローカルのドライブまたはプリンタ	サーバ上のネット名	パスワード
E：	WORD	なし
F：	COMM	fred
PRN：	PRINTE	quick

```
printer      equ   3
drive        equ   4
local_1      db    "e:", 0
local_2      db    "f:", 0
local_3      db    "prin", 0
remote_1     db    "¥¥harold¥word", 0, 0
remote_2     db    "¥¥harold¥comm", 0, "fred", 0
remote_3     db    "¥¥harold¥printer", 0, "quick", 0
func_5F03H: redir local_1, remote_1 drive, 0  ; ドライブを
                                              ;E:WORD という名前で
            jc    error                       ; リダイレクトして渡す
            redir local_2, remote_2 drive, 0  ; ドライブを
                                              ;F:COMM という名前で
            jc    error                       ; リダイレクトして渡す
            redir local_3 remote_3 printer, 0 ; プリンタを
                                              ;PRINTER という名前で
            jc    error                       ; リダイレクトして渡す
```

INT 21H

ファンクション

5F04H 割り当てリストのエントリの取り消し

コール

AH = 5FH
AL = 04H
DS：SI =ソースデバイスの名前の位置

リターン

キャリーフラグがセットされた場合

AX = 01H 無効なファンクションコード。または、MS–Networks が稼働していない
　= 0FH デバイスの停止によるサーバ上のリディレクトの中止

キャリーフラグがセットされない場合

エラーなし

解説

このファンクションは、プリンタまたはディスクドライブ（ソースデバイス）の、ファンクション5FH、コード04Hで作成されたネットワークディレクトリ（ディスティネーションデバイス）へのリダイレクトを取り消します。SIは、取り消すリダイレクトされたプリンタまたはドライブ名を表す、ASCIIZ文字列のオフセットアドレス（DSは、セグメントアドレス）です。MS–Networksの稼動が必要です。

DS：SIで指定されるASCIIZ文字列の値は、次の3つのいずれかです。

1. リダイレクトのコロン付きのドライブ名。リダイレクトを取り消し、物理的なドライブ名に戻す。
2. リダイレクトのプリンタの名前（PRN）。リダイレクトを取り消し、物理的なプリンタ名に戻す。
3. ¥¥（¥マーク2つ）で始まる文字列。ローカルマシンとネットワークディレクトリの接続が終了したことを示す。

マクロ定義

```
cancel_redir    macro   local
                mov     si, offset local
                mov     al, 4
                mov     ah, 5FH
                int     21H
                endm
```

サンプル

次のプログラムは、MS-Networks のドライブ E、F とプリンタ（PRN）のリダイレクトを取り消します。ただし、これらは、ローカルデバイスとして、前もってリダイレクトされていなければなりません。

```
local_1      db         "e:", 0
local_2      db         "f:", 0
local_3      db         "prn", 0
;
func_5F04H: cancel_redir  local_1 ；ドライブ E のリダイレクトを取り消す
            jc            error
            cancel_redir  local_2 ；ドライブ F のリダイレクトを取り消す
            jc            error
            cancel_redir  local_3 ；プリンタ PRN のリダイレクトを取り消す
            jc            error
```

INT 21H

ファンクション
62H　PSPアドレスの取得

コール　AH = 62H

リターン　BX =カレントプロセスのPSPのセグメントアドレス

解　説

このファンクションは、現在実行されているプロセスのセグメントアドレス（PSPの先頭）を返します。

マクロ定義

```
get_psp     macro
            mov     ah, 62H
            int     21H
            endm
```

サンプル　次のプログラムは、PSPのセグメントアドレスを10進数で表示します。

```
msg         db      "PSP segment address:H", 0DH, 0AH, "$"
;
func_62H:   get_psp                        ;PSPのセグメントアドレスを得る
            convert bx, 16, msg[21]        ;章末参照
            display msg                    ;msgを画面に表示（09H）
```

・8086 ファミリーのレジスタ構成

16ビット	上位8ビット	下位8ビット
AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

・フラグレジスタ

ビット	ステータスフラグ（演算結果の状態を表す）
0	CF（キャリーフラグ） 算術演算の結果、最上位ビットが桁上がりをしたとき、セットされます。
2	PF（パリティフラグ） 論理演算の結果、1 になっているビットの個数が偶数のときセット、奇数のときリセットされます。
4	AF（補助キャリーフラグ） 下位ニブルより上位ニブルへ桁上がりしたとき、セットされます。
6	ZF（ゼロフラグ） 演算結果が 0 のときセット、それ以外のときリセットされます。
7	SF（サインフラグ） 演算結果が負のときセット、正のときリセットされます。
11	OF（オーバーフローフラグ） 演算結果が符号も含めてオーバーフローしたとき、セットされます。

ビット	コントロールフラグ（CPU の動作を制御する）
8	TF（トラップフラグ） トラップフラグをセットすると、CPU に命令を 1 ステップだけ実行させることができます。
9	IF（インタラプトフラグ） 外部割り込み要求を受け付けるか否かを指定します。セットすると割り込みを禁止します。
10	DF（ディレクションフラグ） ポインタの内容をインクリメントするかデクリメントするかを指定します。セットするとデクリメント、リセットするとインクリメントします。

1.12　MS-DOS システムコールにおけるマクロ定義例

```
; *******************
; General
; *******************
;
display_asciiz      macro   asciiz_string
                    local   search, found_it
                    mov     bx, offset asciiz_string
;
search:
                    cmp     byte ptr[bx], 0
                    je      found_it
                    inc     bx
                    jmp     short search
;
found_it:
                    mov     byte ptr[bx], "$"
                    display asciiz_string
                    mov     byte ptr[bx], 0
                    display_char 0DH
                    display_char 0AH
                    endm
;
move_string         macro   source, destination, count
                    push    es
                    push    ds
                    pop     es
                    assume  es:code
                    mov     si, offset source
                    mov     di, offset destination
                    mov     cx, count
                    rep movs es:destination, source
                    assume  es:nothing
                    pop     es
                    endm
;
convert     macro   value, base, destination
                    local   table, start
                    jmp     start
                    table   db      "0123456789ABCDEF"
```

```
;
start:
                    push    ax
                    push    bx
                    push    dx
                    mov     al, value
                    xor     ah, ah
                    xor     bx, bx
                    div     base
                    mov     bl, al
                    mov     al, cs:table[bx]
                    mov     destination, al
                    mov     bl, ah
                    mov     al, cs:table[bx]
                    mov     destination[1], al
                    pop     dx
                    pop     bx
                    pop     ax
                    endm
;
convert_to_binary   macro   string, number, value
                    local   ten, start, calc, mult, no_mult
                    jmp     start
                    ten     db      10
;
start:
                    mov     value, 0
                    xor     cx, cx
                    mov     cl, number
                    xor     si, si
;
calc:
                    xor     ax, ax
                    mov     al, string[si]
                    sub     al, 48
                    cmp     cx, 2
                    jl      no_mult
                    push    cx
                    dec     cx
;
mult:
                    mul     cs:ten
```

```
                    loop    mult
                    pop     cx
;
no_mult:
                    add     value, ax
                    inc     si
                    loop    calc
                    endm
;
convert_date        macro   dir_entry
                    mov     dx, word ptr dir_entry[24]
                    mov     cl, 5
                    shr     dl, cl
                    mov     dh, dir_entry[24]
                    and     dh, 1FH
                    xor     cx, cx
                    mov     cl, dir_entry[25]
                    shr     cl, 1
                    add     cx, 1980
                    endm
;
pack_date           macro   date
                    local   set_bit
;
; On entry: DH=day, DL=month, CX=(year-1980)
;
                    sub     cx, 1980
                    push    cx
                    mov     date, dh
                    mov     cl, 5
                    shl     dl, cl
                    pop     cx
                    jnc     set_bit
                    or      cl, 80h
;
set_bit:
                    or      date, dl
                    rol     cl, 1
                    mov     date[1], cl
                    endm
;
```

1.13 MS–DOS システムコールにおける拡張例

```
title DISK DUMP
zero                equ     0
disk_B              equ     1
sectors_per_read    equ     9
cr                  equ     13
blank               equ     32
period              equ     46
tilde               equ     126
        INCLUDE  B:CALLS.EQU
;
subttl DATA SEGMENT
page+
data                segment
;
input_buffer        db      9 dup(512 dup(?))
output_buffer       db      77 dup(" ")
                    db      0DH, 0AH, "$"
start_prompt        db      "Start at sector: $"
sectors_prompt      db      "Number of sectors: $"
continue_prompt     db      "RETURN to continue$"
header              db      "Relative sector$"
end_string          db      0DH, 0AH, 0AH, 07H, "ALL DONE$"

;DELETE THIS
crlf                db      0DH, 0AH, "$"
table               db      "0123456789ABCDEF$"
;
ten                 db      10
sixteen             db      16
;
start_sector        dw      1
sector_num          label   byte
sector_number       dw      0
sectors_to_dump     dw      sectors_per_read
sectors_read        dw      0
;
buffer              label   byte
max_length          db      0
currente_length     db      0
```

```
digits              db      5 dup(?)
;
data                ends
;
subttl STACK SEGMET
page+
stack               segment  stack
dw                  100 dup(?)
stack_top           label    word
stack               ends
;
subttl MACROS
page+
;
        INCLUDE  B:CALLS.MAC
; BLANK LINE
blank_line          macro   number
                    local   print_it
                    push    cx
                    call    clear_line
                    mov     cx, number
print_it:           display output_buffer
                    loop    print_it
                    pop     cx
                    endm
;
subttl ADDRESSABILITY
page+
code                segment
assume              cs:code, ds:data, ss:stack
start               mov     ax, data
                    mov     ds, ax
                    mov     ax, stack
                    mov     ss, ax
                    mov     sp, offset stack_top
;
                    jmp     main_procedure
subttl PROCEDURES
page+
;
; PROCEDURES
; READ_DISK
```

```
read_disk           proc;
                    cmp     sectors_to_dump, zero
                    jle     done
                    mov     bx, offset input_buffer
                    mov     bx, start_sector
                    mov     al, disk_b
                    mov     cx, sectors_per_read
                    cmp     cx, sectors_to_dump
                    jle     get_sector
                    mov     cx, sectors_to_dump
get_sector:         push    cx
                    int     disk_read
                    popf
                    pop     cx
                    sub     sectors_to_dump, cx
                    add     start_sector, cx
                    mov     sectors_read, cx
                    xor     si, si
done:               ret
                    read_disk  endp
;
; CLEAR_LINE
clear_line          proc;
                    push    cx
                    mov     cx, 77
                    xor     bx, bx
move_blank:         mov     output_buffer[bx], ' '
                    inc     bx
                    loop    move_blank
                    pop     cx
                    ret
                    clear_line  endp
;
; PUT_BLANK
put_blank           proc;
                    mov     output_buffer[di], " "
                    inc     di
                    ret
put_blank           endp
;
;
setup               proc;
```

```
                    display     start_prompt
                    get_string  4,  buffer
                    display     crlf
                    convert_to_binary  digits,
                    current_length, start_sector
                    mov         ax, start_sector
                    mov         sector_number, ax
                    display     sectors_prompt
                    get_string 4,  buffer
                    convert_to_binary  digits,
                    current_length, sectors_to_dump
                    ret
                    setup                  endp
;
; CONVERT_LINE
convert_line        proc;
                    push        cx
                    mov         di, 9
                    mov         cx, 16
convert_it:         convert     input_buffer[si], sixteen,
                    output_buffer[di]
                    inc         si
                    add         di, 2
                    call        put_blank
                    loop        convert_it
                    sub         si, 16
                    mov         cx, 16
                    add         di, 4
display_ascii:      mov         output_buffer[di], period
                    cmp         input_buffer[si], blank
                    jl          non_printable
                    cmp         input_buffer[si], tilde
                    jg          non_printable
printable:          mov         dl, input_buffer[si]
                    mov         output_buffer[di], dl
non_printable:      inc         si
                    inc         di
                    loop        display_ascii
                    pop         cx
                    ret
                    convert_line        endp
;
```

```
; DISPLAY_SCREEN
display_screen      proc;
                    push        cx
                    call        clear_line
;
                    mov         cx, 17
; I WANT length header
                    dec         cx
; minus 1 in cx
                    xor         di, di
move_header:        mov         al, header[di]
                    mov         output_buffer[di], al
                    inc         di
                    loop        move_header     ; FIX THIS!
;
                    convert     sector_num[1], sixteen
                    output_buffer[di]
                    add         di, 2
                    convert     sector_num, sixteen,
                    output_buffer[di]
                    display     output_buffer
                    blank_line 2
                    mov         cx, 16
dump_it:            call        clear_line
                    call        convert_line
                    display     output_buffer
                    loop        dump_it
                    blank_line 3
                    display     continue_prompt
                    get_char_no_echo
                    display     crlf
                    pop         cx
                    ret
                    display_screen      endp
;
;
; END PROCEDURES
subttl MAIN PROCEDURE
page+
main_procedure:     call        setup
check_done:         cmp         sectors_to_dump, zero
                    jng         all_done
```

```
                call        read_disk
                mov         cx, sectors_read
display_it:     call        display_screen
                call        display_screen
                inc         sector_number
                loop        display_it
                jmp         check_done
all_done:       display     end_string
                get_char_no_echo
code            ends
                end         start
```

第2章

拡張機能

2.1 イントロダクション

PC-9800シリーズの本体のハードウェア資源を、有効に利用するために、いくつかの拡張機能がプログラムで操作できるようになっています。

ここでは、これらの拡張機能を解説します。なお、ノーマルモードとハイレゾリューションモードで動作や操作に違いがある場合は、原則としてノーマルモードでの解説を基にして、ハイレゾリューションモードにおける違いを解説します。

2.2 拡張機能の利用方法

拡張機能を呼び出すときは、CLレジスタに機能コードを格納し、その他の必要な情報を各レジスタに設定して、割り込みタイプDCH（INT DCH）を実行します。

呼び出し後のレジスタの内容は、リターンで定義されているレジスタ以外はすべて保障されます。機能が定義されていない機能コードを呼び出しても何も実行されません。

2.3 拡張機能呼び出し

PC-9800シリーズでは、次のような機能が拡張機能として用意されています。

機能コード（16進）	機　能
0AH	RS-232Cポートの初期化
0CH	キーの取得
0DH	キーの設定
0EH	RS-232Cポートの操作
0FH	CTRL＋ファンクションキーのソフトキー化／解除
10H	直接コンソール出力
11H	プリンタモードの変更

このうち、RS-232C に関する 0AH と 0EH を利用する場合、デバイスドライバの〝RSDRV.SYS〟をシステムに組み込んでおく必要があります。また、同様に 11H を利用するときは、〝PRINT.SYS〟を組み込んでおかなければなりません。組み込み方法については、ユーザーズリファレンスマニュアルを参照してください。

ここでは、各拡張機能呼び出しごとに解説を行います。

機能コード **0AH**

RS–232Cポートの初期化

コール

CL ＝0AH
DX ＝パラメータ（下表参照）

リターン

AX ＝0　　　正常終了
　 ＝FFFFH 異常終了

DH ＝

7	6	5	4	3	2	1	0	機能	
							0 1	Xパラメータ	無効 有効
						0			
				1 1	0 1			データビット長	7ビット 8ビット
			0 1					パリティチェック	なし あり
		0 1						パリティ指定	奇数 偶数
0 1	1 1							ストップビット長	1ビット 2ビット

（上表の見出し：データ（ビットの位置））

DL ＝

7	6	5	4	3	2	1	0	機能	
				0 0 0 0 0 0 0 0 1	0 0 0 0 1 1 1 1 0	0 0 1 1 0 0 1 1 0	0 1 0 1 0 1 0 1 0	ボーレート	無効 75BPS 150BPS 300BPS 600BPS 1200BPS 2400BPS 4800BPS 9600BPS
0 0 0	0 0 0	0 0 1	0 1 0					チャンネル番号	0 1 2

（上表の見出し：データ（ビットの位置））

解　　説

レジスタ DX にセットされたチャンネル番号の RS–232C ポートを初期設定します。

この機能は、PC–98XL/XA などのハイレゾリューションモードでのみ使用可能です。ノーマルモードでは、機能コード 0EH を利用してください。機能コード 0EH は、ノーマルモード、ハイレゾリューションモード共通に使用できます。

1 デフォルトはチャンネル 0 で、チャンネル 1 および 2 については、拡張ボードがセットされていなければ初期化は行われません。また、レジスタ DL でセットするボーレートはチャンネル 0 についてのみ有効です。チャンネル 1 および 2 は拡張ボード上のスイッチにより× 16 モードでセットしてください。

機能コード
0CH　キーの取得

コール

CL ＝0CH
DX ＝データバッファのオフセット
DS ＝データバッファのセグメント
AX ＝0000H　ノーマルモード全ソフトキーを取得
　＝00FFH　ハイレゾリューションモード全ソフトキーを取得

　＝0001～000AH　f･1 ～ f･10 の取得
　＝000B～0014H　SHIFT ＋ f･1 ～ SHIFT ＋ f･10 の取得
　＝0015～001FH　カーソル移動キーなどの取得（*）
　＝0020～0024H　f･11 ～ f･15 の取得
　＝0025～0029H　SHIFT ＋ f･11 ～ SHIFT ＋ f･15 の取得
　＝002A～0038H　CTRL ＋ f･1 ～ CTRL ＋ f･15 の取得

　＝0100H　データキー割り当てバッファの内容の取得
　＝0101H　データキー割り当てバッファの残りサイズの取得
　（AX＝0101Hのときは、レジスタDS、DXにバッファアドレスをセットする必要なし）

（*）

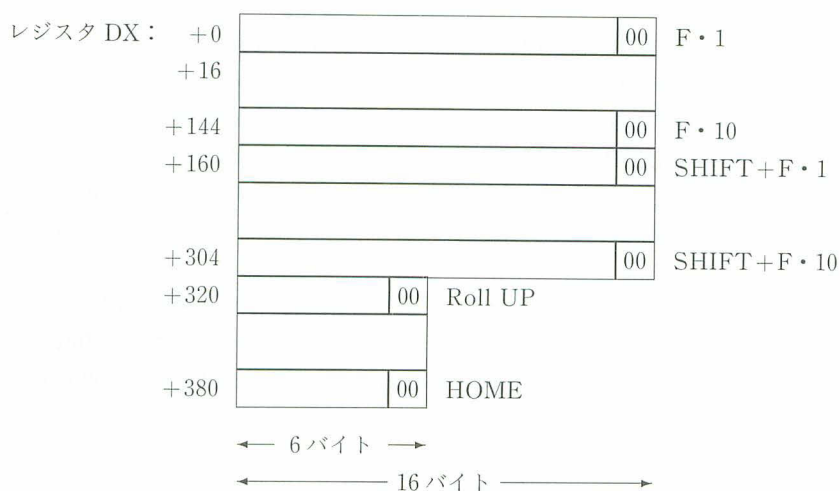
AX＝				
0015H	ROLL UP	0016H	ROLL DOWN	
0017H	INS	0018H	DEL	
0019H	↑	001AH	←	
001BH	→	001CH	↓	
001DH	ノーマルモード時			HOME CLR
	ハイレゾリューションモード時			CLR
001EH	HELP			
001FH	ノーマルモード時			SHIFT ＋ HOME CLR
	ハイレゾリューションモード時			HOME

解　説

ファンクションキーやカーソル移動キーなどの取得を行います。レジスタ DX にセットしたアドレスのバッファに、ファンクションキーやカーソル移動キーに現在割り当てられている機能を書き込みます。

レジスタ AX に 0100H をセットしてこの機能を呼び出すと、レジスタ DX にセットしたアドレスのバッファに、データキーに現在再割り当てされている機能を書き込みます。

レジスタ AX に 0101H をセットしてこの機能を呼び出すと、データキーに機能を再割り当てするための内部バッファの残りのバイト数をレジスタ AX に返します。

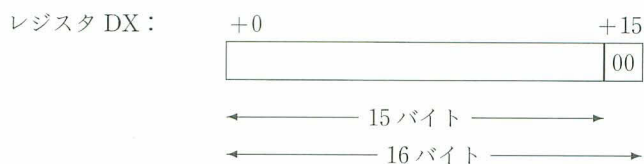
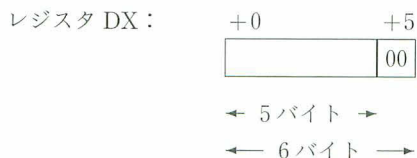
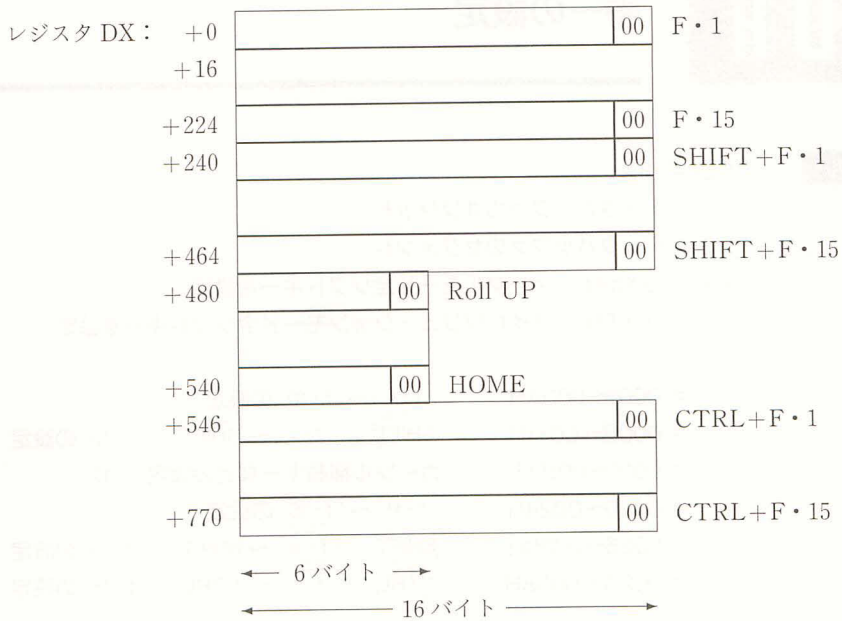
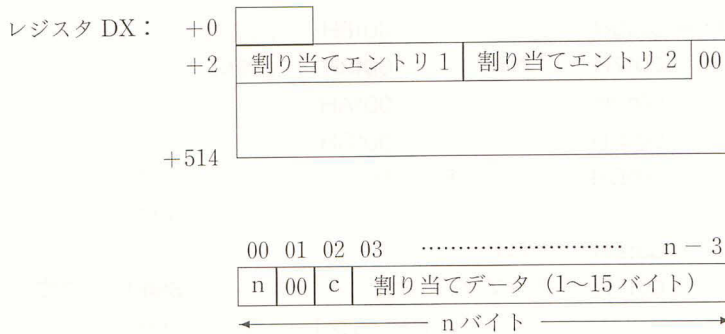
バッファの形式

AX = 0001H〜0014H あるいは 0020H〜0038H の場合

AX = 0015H〜001FH の場合

レジスタ DX：　+0　+5

00

5 バイト

6 バイト

AX = 00FFH の場合

AX = 0100H の場合

n…………割り当てエントリ全体の長さ（バイト数）
c…………割り当てるキーコード（英数、英記号、カナ、カナ記号、20H〜7EH、A0H〜DFH）

オフセット 00 から 1 ワードには、割り当てエントリの数が格納されています。
オフセット 02 から 512 バイトはバッファとして使用され、割り当てエントリが格納されます。
下側の図は、1 つの割り当てエントリの形式を表したものです。

機能コード

0DH キーの設定

コール

CL ＝0DH
DX ＝データバッファのオフセット
DS ＝データバッファのセグメント
AX ＝0000H　ノーマルモード全ソフトキーを設定
　 ＝00FFH　ハイレゾリューションモード全ソフトキーを設定

＝0001〜000AH　[f･1]〜[f･10]の設定
＝000B〜0014H　[SHIFT]＋[f･1]〜[SHIFT]＋[f･10]の設定
＝0015〜001FH　カーソル移動キーなどの設定（*1）
＝0020〜0024H　[f･11]〜[f･15]の設定
＝0025〜0029H　[SHIFT]＋[f･11]〜[SHIFT]＋[f･15]の設定
＝002A〜0038H　[CTRL]＋[f･1]〜[CTRL]＋[f･15]の設定

＝0100H　データキー割り当てバッファの内容設定（*2）
＝0101H　データキー割り当てバッファの割り当てエントリを１つ追加

（*1）

AX＝			
0015H	[ROLL UP]	0016H	[ROLL DOWN]
0017H	[INS]	0018H	[DEL]
0019H	[↑]	001AH	[←]
001BH	[→]	001CH	[↓]
001DH	ノーマルモード時		[HOME CLR]
	ハイレゾリューションモード時		[CLR]
001EH	[HELP]		
001FH	ノーマルモード時		[SHIFT]＋[HOME CLR]
	ハイレゾリューションモード時		[HOME]

（*2）

データキー割り当てバッファの内容は、機能コード 0CH を参照してください。このコールではオフセット０からの１ワードにエントリ数を格納する必要はありません。

解　　説

ファンクションキーやカーソル移動キーなどの設定を行います。レジスタ DX にアドレスが格納されているバッファの機能をファンクションキーやカーソル移動キーに割り当てます。

レジスタ AX0000H のときは、ノーマルモードで使用可能なファンクションキー、カーソル移動キーすべてに機能を設定します。

レジスタ AX00FFH のときは、ハイレゾリューションモードで使用可能なファンクションキー、カーソル移動キーすべてに機能を設定します。

データバッファの形式は、機能コード 0CH と同じです。

各キーに対する有効文字列の最後には、00H が 16 バイトあるいは 6 バイトを満たすまでおぎなっておく必要があります。

レジスタ AX に 0100H をセットしてこの機能呼び出しを行うと、レジスタ DX でアドレスを指定したバッファ内のデータキー割り当て情報をシステムの内部バッファに設定します。

また、レジスタ AX に 0101H をセットしてこの機能呼び出しを行うと、データキーへの機能割り当てを内部バッファに 1 エントリだけ追加します。

割り当てエントリの形式は機能コード 0CH を参照してください。

注意　CTRL ＋ f･1 ～ CTRL ＋ f･15 に割り当てた機能を実際に使用するためには、機能コード 0FH の AX ＝ 0000H の呼び出しによって CTRL ＋ f･1 ～ CTRL ＋ f･15 をソフトキー化しなければなりません。

RS–232C ポートの操作

コール

CL ＝ 0EH
DL ＝パラメータ（下表参照）
　　00H ＝チャンネル０の受信データ通知
　　10H ＝チャンネル１の受信データ通知
　　20H ＝チャンネル２の受信データ通知
　　01H ＝チャンネル０の初期設定
　　11H ＝チャンネル１の初期設定
　　21H ＝チャンネル２の初期設定

DL ＝ 01H、11H、21（初期設定）のとき
BX ＝パラメータ（下表参照）

BH ＝

データ（ビットの位置）								機　能	
7	6	5	4	3	2	1	0		
							0 1	X パラメータ	無効 有効
						0			
				1 1	0 1			データビット長	7 ビット 8 ビット
			0 1					パリティチェック	なし あり
		0 1						パリティ指定	奇数 偶数
0 1	1 1							ストップビット長	1 ビット 2 ビット

BL ＝

データ（ビットの位置）								機　能	
7	6	5	4	3	2	1	0		
				0 0 0 0 0 0 0 0 1	0 0 0 0 1 1 1 1 0	0 0 1 1 0 0 1 1 0	0 1 0 1 0 1 0 1 0	ボーレート	無効 75BPS 150BPS 300BPS 600BPS 1200BPS 2400BPS 4800BPS 9600BPS
0	0	0	0						

リターン

DL ＝00H、10H、20H（データ長取得）のとき
AX ＝受信データのデータ長

DL ＝01H、11H、21H（初期設定）のとき
AX ＝0000H　正常終了
　＝FFFFH　異常終了（拡張 RS–232C ボードが実装されていません）

解　説

指定されたチャンネル番号の RS–232C ポートの初期設定または受信データ長の通知を行います。

DL＝x0H（x は 0〜2）のとき、指定された RS–232C ポートに受信しているデータ長を、レジスタ AX に返します。

DL＝x1H（x は 0〜2）のとき、レジスタ BX にセットされたパラメータで RS–232C ポートが初期設定されます。

チャンネル 1 および 2 については、拡張 RS–232C ボードが本体に実装されていなければ初期設定は行われません。また、レジスタ BL にセットされるボーレートはチャンネル 0 のみ有効です。チャンネル 1 および 2 は拡張 RS–232C ボード上のスイッチにより× 16 モードでセットしてください。

機能コード 0FH　CTRL＋ファンクションキーのソフトキー化／解除

コール

CL ＝0FH
AX ＝0000H　CTRL＋ファンクションキーのソフトキー化
　 ＝0001H　CTRL＋ファンクションキーのソフトキー解除

解説

CTRL＋ファンクションキーのソフトキー化およびその解除を行います。
ノーマルモードの［CTRL］＋［f･1］～［f･10］、ハイレゾリューションモードでの［CTRL］＋［f･1］～［f･15］は、通常のMS–DOSでは直接コンソール入出力（第１章「ファンクションリクエスト06H」参照）では正常なキー値を得ることができません。しかし、レジスタAXに0000Hをセットし、この機能呼び出しを行うことで正常なキー値を得ることができるようになります。

［CTRL］＋［f･1］～［f･10］または［CTRL］＋［f･1］～［f･15］の扱いを通常のMS–DOSに戻すには、レジスタAXに0001Hをセットしてこの機能呼び出しを行ってください。

KEYコマンドで設定したCTRL＋ファンクションキーの機能を利用するためには、レジスタAXに0000Hを入れて、この機能呼び出しを行わなければなりません。

機能コード
10H 直接コンソール出力

コール

CL　＝10H
AH　＝サブファンクション番号（00–0EH）
その他＝サブファンクションごとに必要なレジスタ

解　　説

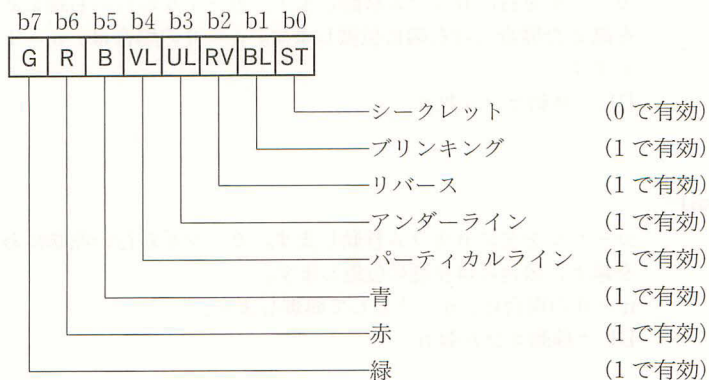
AHにセットされたサブファンクション番号に応じて、ディスプレイ画面に対して直接出力動作を行います。サブファンクションごとの機能およびコール条件は次のとおりです。

AH＝00H
機能：　ディスプレイ画面、1バイトのデータを出力します。漢字を出力するにはシフトJISコードの第1バイト、第2バイトを順に出力してください。
コール：　DL＝出力するキャラクタ

AH＝01H
機能：　ディスプレイ画面に文字列を出力します。文字列の終わりには‘$’をセットしてください。
コール：　DS：DX＝文字列の先頭アドレス

AH＝02H
機能：　文字の属性を変更します。このサブファンクション実行後は、設定した属性が以後に続く文字に対して適用され、次の変更まで有効です。
コール：　DL＝文字の属性

b7	b6	b5	b4	b3	b2	b1	b0
G	R	B	VL	UL	RV	BL	ST

- ST：シークレット　(0で有効)
- BL：ブリンキング　(1で有効)
- RV：リバース　(1で有効)
- UL：アンダーライン　(1で有効)
- VL：バーティカルライン　(1で有効)
- B：青　(1で有効)
- R：赤　(1で有効)
- G：緑　(1で有効)

AH＝03H
機能：　カーソル位置の設定を行います。
コール：　DH＝ライン
DL＝カラム

AH＝04H
機能：　カーソルを同じカラムで下に１行移動します。カーソルが最終行にある場合は１行スクロールアップします。
コール：　なし

AH＝05H
機能：　カーソルを同じカラムで上に１行移動します。カーソルが先頭行にある場合は１行スクロールダウンします。
コール：　なし

AH＝06H
機能：　カーソルを同じカラムで上にｎ行移動します。カーソルが先頭行にあるか、先頭行を越えた場合には、先頭行に位置します。ｎ＝０の場合は、ｎ＝１として処理します。
コール：　DL＝移動行数ｎ

AH＝07H
機能：　カーソルを同じカラムで下にｎ行移動します。カーソルが最終行にあるか、最終行を越えた場合には、最終行に位置します。ｎ＝０の場合は、ｎ＝１として処理します。
コール：　DL＝移動行数ｎ

AH＝08H
機能：　カーソルを右にｎカラム移動します。カーソルが行の右端にあるか、右端を越えた場合には右端に位置します。ｎ＝０の場合は、ｎ＝１として処理します。
コール：　DL＝移動カラム数ｎ

AH＝09H
機能：　カーソルを左にｎカラム移動します。カーソルが行の左端にあるか、左端を越えた場合には左端に位置します。
ｎ＝０の場合は、ｎ＝１として処理します。
コール：　DL＝移動カラム数ｎ

AH＝0AH
機能：　ディスプレイ画面のクリアをコール条件にしたがって行います。

コール：　DL　＝00H　カーソル位置から最終行右端までをクリアします。
　＝01H　先頭行左端からカーソル位置までをクリアします。
　＝02H　画面全体をクリアします。
　これらの値以外は無視されます。

AH＝0BH
機能：　現在行のクリアをコール条件にしたがって行います。

コール：　DL　＝00H　カーソル位置から行の右端までをクリアします。
　＝01H　行の左端からカーソル位置までをクリアします。
　＝02H　カーソルの位置する行を左端から右端までクリアします。

AH＝0CH
機能：　カーソルの位置する行以降をn行下に移動し、空白のn行を挿入します。カーソルは先頭の挿入行の左端に位置します。挿入行が最終行を越えた場合、移動する行が最終行を越えた場合は、その越えた行は失われます。
n＝0の場合は、n＝1として処理します。
コール：　DL＝挿入する行数n

AH＝0DH
機能：　カーソルの位置する行から下にn行削除し、以降の行を上に詰めます。カーソルの位置は詰められた行の左端になります。最終行を越えての削除は行われません。
n＝0の場合は、n＝1として処理します。
コール：　DL＝削除する行数n

AH＝0EH
機能：　81～9FHあるいはE0～FCHまでのコードをシフトJISコードの第1バイトとして扱うか、グラフ文字として扱うかのモード指定を行います。

コール：　DL　＝00H　シフトJISモード（システムの既定値）
　＝03H　グラフ文字モード
　これらの値以外は無視されます。

10H

機能コード
11H プリンタモードの変更

コール

CL ＝11H
AX ＝0000H　ドットスペイシングを行わないモードにする
　＝0001H　ドットスペイシングを行うモードにする
　＝0020H　偶数回目の CTRL ＋ P でプリンタに CR/LF コードを出力しない
　＝0021H　偶数回目の CTRL ＋ P でプリンタに CR/LF コードを出力する

解　　説

プリンタのモードを制御して、ANK（1バイト系英数カナ）文字と漢字の大きさの比率を変更することができます。

AX ＝0000H　日本語プリンタにおいて、ANK文字と漢字の比率が1：1.5になります。
　＝0001H　日本語プリンタにおいて、ANK文字と漢字の比率が1：2になります。
　＝0020H　偶数回目の CTRL ＋ P を押した（画面出力のプリンタへのエコー解除）ときに、プリンタにCR/LFコードを出力しないモードとなります。レジスタAXに0021Hをセットした場合は、この逆にCR/LFコードを出力するようになります。

注意　プリンタを使用するためにはデバイスドライバ〝PRINT.SYS〟が組み込まれていなければなりません。

第3章

MS–DOS 技術資料

3.1 MS–DOSの初期化

MS–DOSの初期化は、次のようなステップで行われます。まず、ROM（リードオンリーメモリ）内のブートストラップに制御が渡され、次に、このブートストラップによってディスクからブートセクタが読み込まれます。続いてこのブートセクタによって、次のファイルが読み込まれます。

IO.SYS
MSDOS.SYS

これらのファイルが読み込まれると、ブート処理を開始します。

3.2 コマンドプロセッサ

MS–DOSコマンドプロセッサ（COMMAND.COM）は、常駐部、初期化部、非常駐部の3つの部分から構成されています。

1. 常駐部は、MSDOS.SYSと、そのデータ域のすぐ後のメモリ上に配置されます。この部分は、割り込みタイプ22H（終了アドレス）、23H（<CTRL–C>抜け出しアドレス）、24H（致命的なエラーによる打ち切りアドレス）を処理するためのルーチン、および必要に応じて、非常駐部をロードするためのルーチンから構成されています（プログラムの終了時、チェックサム方式によってプログラムが非常駐部にオーバーレイが行われたか調べます。オーバーレイが行われた場合は再ロードを行います）。すべての標準MS–DOSエラーハンドリングは、COMMAND.COMのこの部分で行われます。このハンドラには、エラーメッセージの画面出力および〝中止<A>, もう一度<R>, 無視<I>?〞の応答の解読ルーチンが含まれています。

2. 初期化部は常駐部の次に存在し、開始時に制御が渡されます。この部分にはAUTOEXEC.BATファイルの処理ルーチンが入っています。プログラムのロード可能なセグメントアドレスは、初期化部分によって決定されます。それ以後は必要ないため、最初にロードされるCOMMAND.COMのプログラムによってオーバーレイされます。

3. 非常駐部は、メモリの最上位にロードされます。この部分には、すべての内部コマンドとバッチファイルプロセッサが入っています。

コマンドプロセッサの3番目の部分によって、プロンプト（A>のような）が表示されコマンドのキーボード（またはバッチファイル）入力と実行が行われます。外部コマンドの場合、コマンドラインを作成し、プログラムのロードと制御の移行を行うためのEXECファンクションコール（ファンクション4BH、コード00H）が行われます。

3.3 MS-DOS ディスクアロケーション

MS-DOSのディスクスペースは、次のようなフォーマットになっています。領域のサイズはいずれも可変です。

予備領域
ファイルアロケーションテーブル（FAT）1
ファイルアロケーションテーブル（FAT）2
ルートディレクトリ
データ領域

ファイルのためのスペース（データ領域）は、必要に応じて割り当てられます。前もって割り当てられるものではありません。スペースは、一度に1クラスタ（アロケーションの単位）ずつ割り当てられます。クラスタとは、常に連続したいくつかのセクタのことで、クラスタは、ファイルアロケーションテーブル（FAT）を通して〝連結〟されています。1クラスタの中のセクタ数は必ず2の累乗です。また、信頼性を高めるために、最初のFATをバックアップした2番目のFATが保存されています。第1のFATの途中にスキップセクタが発生して管理情報が失われた場合でも、2番目のFATを使用することができ、使用不可になったディスクでもデータを回復することができます。

3.4 MS-DOS ディスクディレクトリ

FORMATコマンドは、すべてのディスクにルートディレクトリを作成します。ディレクトリのロケーション（論理セクタ番号）および最大のエントリ数は、メディアによって決まります。

ルートディレクトリ以外のディレクトリはファイルと同じなので、無制限に作成することができます。

ディレクトリの長さは32バイトで、次のようなフォーマットで記入されます（オフセットは16進）。

オフセット		サイズ（バイト）	内容
16進	10進		
00H～07H	0～7	8	ファイル名
08H～0AH	8～10	3	拡張子
0BH	11	1	属性
0CH～15H	12～21	10	予約エリア
16H～17H	22～23	2	最終編集時刻 ビット　0～4＝秒/2 5～10＝分 11～15＝時
18H～19H	24～25	2	最終編集日付 ビット　0～4＝日 5～8＝月 9～15＝年
1AH～1BH	26～27	2	開始クラスタ
1CH～1FH	28～31	4	ファイルサイズ，バイト単位

00～07H　8文字のファイル名。8文字に満たない場合は左詰めで、残りにスペースが入ります。また、このフィールドの先頭バイトは次のようなステータスを示します。

00H　未使用。性能上の理由から、ディレクトリ検索の長さを制限するためのもの。

05H　ファイル名の先頭の1文字が実際にはE5Hであることを示す。

E5H　すでに消去されたファイル。

2EH　これはディレクトリのためのもの。2バイト目も2EHの場合、クラスタフィールドには、このディレクトリの親ディレクトリのクラスタ番号が入っている(親ディレクトリがルートディレクトリの場合は0000H)。

上記以外の文字の場合、ファイル名の先頭の文字になります。

08～0AH　ファイル名拡張子

0BH　ファイルのアトリビュート（属性）。アトリビュートバイトは、次のようにマップされます（値は16進）。

01H　書き込み不可。このファイルをファンクション3DHで、書き込みのためにオープンしようとしてもエラーコードが返される。また、ファイルの削除(13H)、ディレクトリの削除（41H）もエラーになる。この値は、以下の他の値と一緒に使用することができる。

02H　隠されたファイル。このファイルは、通常のディレクトリ検索から除外される。

04H　システムファイル。このファイルは、通常のディレクトリ検索から除外される。

08H　このエントリの最初の 11 バイトには、ボリュームラベルが入っている。このエントリは、作成の日時以外には一般的な情報が入っておらず、ルートディレクトリにのみ存在することができる。

10H　このエントリはサブディレクトリを定義し、通常のディレクトリ検索から除外される。

20H　保存ビット。このビットは、ファイルが新規に作成された、更新されたときにオンにセットされる。このビットは、他のアトリビュートビットと一緒に使用することができる。

注意　IO.SYS、MSDOS.SYS には、リードオンリー、隠されたファイル、システムファイルのマークが付けられます。ファイルは作成時に、隠されて見えないファイルのマークを付けることができます。またリードオンリー、隠されたファイル、システムおよび保存の属性は、ファンクション 43H によって変更可能です。

0C～15H　予約域

16～17H　ファイルが作成された時刻または最後に編集された時刻が、次のようなビット列にマップされます（左がビット 15、右がビット 0 です）。

オフセット 17H								オフセット 16H							
15				11	10					5	4				0
H	H	H	H	H	M	M	M	M	M	M	S	S	S	S	S
時					分						秒/2				

HHHHH　　：時　2 進数で表した時刻（0～23）
MMMMMM　：分　2 進数で表した分（0～59）
SSSSS　　：秒　秒/2

18～19H　ファイルが作成されたときまたは最後に編集された日付。
年／月／日は、次のようなビット列にマップされます。

オフセット 19H								オフセット 18H							
15						9	8			5	4				0
Y	Y	Y	Y	Y	Y	Y	M	M	M	M	D	D	D	D	D
年							月				日				

MMMM　　：月　1～12
DDDDD　　：日　1～31
YYYYYYY　：年　0～99（1980～2079）

参考　MS–DOS は 1980 年を基点に年度を設定しています。

1A～1BH　開始クラスタ。ファイルの先頭クラスタの相対クラスタ番号。すべてのディスクのデータスペースの先頭のクラスタは、クラスタ 002 となる。クラスタ番号は、最下位バイトから先に記憶される。

注意　クラスタ番号を論理セクタ番号に変換する場合の詳細については、3.5「MS–DOS ファイルアロケーションテーブル」を参照してください。

1C～1FH　バイトで表したファイルの大きさ。最初のワードには、ファイルの大きさの下位の部分が入っている。両方のワードとも、下位のバイトから先に記憶される。

3.5　MS–DOS ファイルアロケーションテーブル

本章は、デバイスドライバを開発するシステムプログラムのための解説です。MS–DOS で使用しているファイルアロケーションテーブル（FAT）が、どのようにクラスタを論理セクタ番号に変換するか解説します。

参考　フロッピィディスクなどの円盤型のメディアでは、トラックと呼ばれる年輪状に区切られた円周上を、さらに分割したセクタと呼ばれる部分に情報は記録されますが、同一トラック上のセクタはクラスタと呼ばれるひとつの単位として扱われます。ファイルアロケーションテーブルには、どのファイルをどのクラスタに書き込んだかという配置情報が記録されています。

ディスク上の論理セクタの位置決めは、ドライバが行います。この情報は、ドライバ以外の方法でアクセスすべきではありません。システムユーティリティプログラムは、FAT に直接アクセスするのではなく、MS–DOS ファイル管理ファンクションコールを使用すべきです。

FAT は、通常各クラスタごとに 12 ビットのエントリで作成されます。ただし、固定ディスクなど、クラスタ数の最大値が 4085 を超えるような種類のディスクでは、16 ビットのエントリで作成されます。12 ビットのエントリの場合、先頭の 2 つの FAT のエントリは、ディレクトリの一部を示しており、これらの FAT にはディスクの大きさとフォーマットを示す標識が入っています。2 バイト目と 3 バイト目には、常に FFH が入っています。

3 番目の FAT から、データ領域のマッピングが始まります（クラスタ 002）。各エントリにも、16 進で表した 3 文字（16 ビットエントリの場合は 4 文字）が入っており、それぞれ次のような内容を示します。

(0)000H　クラスタは未使用で、使用可能。

(F)FF7H　クラスタに、スキップセクタ（不良セクタ）が入っている。MS–DOS は、このようなクラスタは割り当てない。このクラスタ数が CHKDSK によって数えられ、通知される。

(F)FF8～(F)FFFH　ファイル内の最終クラスタを示している。

(X)XXXH　上記以外の 16 進数の場合、ファイル内の次のクラスタのクラスタ番号を示している。ファイルの先頭のクラスタ番号は、ディレクトリエントリに保存される。

ディスクには通常、信頼性を高めるためにFATは2つ作られています。FATは必要なとき（ファイルのオープン、これ以上のスペースを割り当てるなど）、MS-DOSバッファの1つに読み込まれます。性能上の理由から、このバッファには高いプライオリティ（優先順位）が与えられ、可能なかぎり長くメモリ内に保存されます。

■12ビットFATエントリ

まずディレクトリエントリから、ファイルの開始クラスタの番号を取得します。ファイルの次に来るクラスタの位置を指定する場合、次のことを行います。

1. 現在使用されているクラスタ番号を1.5倍にします（各FATエントリは、1.5バイトの長さです）。

2. この積全体がFAT内のオフセットで、現在使用されているクラスタをマップするエントリを指します。このエントリには、ファイル内の次のクラスタのクラスタ番号が入っています。

3. 計算されたFATオフセットにある1ワードをレジスタに入れるため、MOV命令を使用します。

4. 使用された最終クラスタが偶数の場合、このレジスタの内容にFFFHを加算することによってこのレジスタの下位12ビットを保存するか、またはSHR命令を使用してこのレジスタの内容を右に4ビットシフトして上位12ビットを保存してください。

5. 結果として取得された12ビットがFF8HからFFFHまでの値を取る場合、ファイル内にこれ以上のクラスタは存在しません。これ以外の値であると、この12ビットには、ファイル内の次のクラスタのクラスタ番号が入っています。

このクラスタを論理セクタ番号（割り込みタイプ25Hと26HおよびSYMDEBによって使用されるような、相対セクタ）に変換する場合、次のことを行ってください。

1. クラスタ番号から2を引く。
2. この演算結果に1クラスタ当りのセクタ数を掛ける。
3. データ領域内の開始論理セクタ番号を加える。

■16 ビット FAT エントリ

まずディレクトリエントリから、ファイルの開始クラスタの番号を取得します。次のファイルのクラスタの位置をしていする場合、次のことを行います。

1. 現在使用されているクラスタ番号を 2 倍にします（各 FAT エントリは、2 バイトの長さです）。

2. 計算された FAT オフセットにある 1 ワードをレジスタ内に入れるため、MOV WORD 命令を使用します。

3. 結果として取得された 16 ビットが FFF8H から FFFFH までの値を取る場合、ファイル内にこれ以上のクラスタは存在しません。これ以外の値の場合、この 16 ビットには、ファイル内の次のクラスタのクラスタ番号が入っています。

第4章

MS-DOSコントロールブロックとワークエリア

4.1 MS–DOS メモリマップ

XXXX：0000	割り込みのベクタテーブル
XXXX：0000	IO.SYS MS–DOS とハードウェアのインターフェイス
XXXX：0000	MSDOS.SYS MS–DOS 割り込みハンドラ、サービスルーチン（割り込みタイプ 21H)、MS–DOS バッファ、コントロールエリアおよび登録されているデバイスドライバ
XXXX：0000	COMMAND.COM の常駐部 割り込みタイプ 22H（終了アドレス）、23H（<CTRL–C>による抜け出しアドレス)、24H（致命的エラーによる打ち切りアドレス）のための割り込みハンドラおよび非常駐部分をロードし直すためのコード
XXXX：0000	外部コマンドまたはユーティリティ（.COM、.EXE ファイル）
XXXX：0000	.COM ファイルのためのユーザースタック（256 バイト）
XXXX：0000	COMMAND.COM の非常駐部 コマンドプロセッサ、内部コマンド、バッチプロセッサ

ユーザーメモリは、メモリに対するリクエストの条件を満たす、使用可能な一番低いメモリの終わりから割り当てられます。

4.2 MS–DOS プログラムセグメント

外部コマンドを入力した場合、または EXEC ファンクションコールによってプログラムをコールした場合、MS–DOS は、使用可能な最下位のアドレスを、コマンドやプログラムのためのメモリの開始アドレスとします（ただし、EXE 形式のファイルの minalloc と maxalloc がともにゼロであると、ファイルは可能な限り高いアドレスへロードされます)。

プログラム開始アドレスからの 256 バイトは、プログラムがロードされたとき、EXEC システムコールによってセットアップされます。この領域を PSP（プログラムセグメントプレフィクス）と呼びます。プログラムは、このブロックの次にロードされます。

■ プログラムセグメントプレフィックス（PSP）のフォーマット

PSP は次のようなフォーマットになっています。

オフセット				
0H	INT20H	使用可能な最初のセグメント①	リザーブ	MS-DOS機能をロングコールするための5バイト(オフセットアドレス)②
8H	MS-DOSをロングコールするためのセグメントアドレス	終了アドレス（IP、CS）		<CTRL–C>の抜け出しアドレス（IP）
10H	<CTRL–C>の抜け出しアドレス（CS）	ハードエラーによる抜け出しアドレス（IP、CS）		
	16H～5BH リザーブ（MS-DOS が使用可能）	2CH		
	5CH　パラメータ 1　（通常はオープンされていない FCB）			
	6CH　パラメータ 2　〔通常はオープンされていない FCB、5CH の FCB がオープンされていると、オーバーライトされる〕			
80H	パラメータ 3　（通常は DTA ③）初期化されたコマンドインボケーションライン			
100H				

注意　①使用可能なメモリ上の最初のセグメントは、セグメント（パラグラフ）のフォーマットで表します（たとえば、1000H は 64K を表します）。
②オフセット 06H にある 1 ワードには、セグメント内で使用可能なバイト数が入っています。
③デフォルトの DTA として 80H～FFH を使用できます。ただし、DTA として利用するとパラメータは破壊されます。

重要　PSP のオフセット 5CH 未満の部分は、プログラムによって変更しないでください。

EXEC で起動されたプログラムを元に戻す場合、次の 5 つのいずれかの方法を使います。

1. AH = 4CH で INT21H を行う
2. AH = 31H で INT21H を行う（KEEP PROCESS）
3. PSP 内のオフセット 0 に long ジャンプを行う
4. INT20H を行う（CS：0 は PSP を指していなければいけません）
5. AH = 0 で INT21H を行う（CS：0 は PSP を指していること）

注意　機能的に将来の MS–DOS のバージョンに対応しやすいため、1 または 2 の方法を使用するのが望ましいでしょう。

5 つの方法のいずれを使用した場合でも、結果として EXEC のコールを行ったプログラムに制御が渡されます。ただし、1 と 2 の方法は戻るときの終了コードを指定できます。戻るとき、割り込みベクタ 22H、23H、24H（終了アドレス、<CTRL–C >抜け出しアドレス、致命的エラーによる打ち切りアドレス）のアドレスが、終了したプログラムのプログラムセグメントプレフィックス内に保存されていた値により回復します。こうして次に、制御が終了アドレスに渡されます。COMMAND に戻るプログラムの場合、制御は COMMAND の常駐部に渡され、バッチファイルを処理中のときは、これを続行します。それ以外の場合、COMMAND によって非常駐部に対するチェックサムが行われ、必要な場合再ロードが行われます。次に COMMAND はシステムプロンプトを出力し、キーボードからの次の入力を待ちます。

以下、PSP について詳細に解説します。

・オフセット 2CH

渡された環境のセグメントアドレスは、PSP のオフセット 2CH に入っています。この環境とは、次のようなフォーマットによる一連の ASCII 文字列（合計が 32K 未満）のことです。

環境変数名＝パラメータ

各文字列は、1 バイトのゼロによって区切られ、文字列全体は、さらに 1 バイトのゼロが続くことによって終了します。その最後のゼロに続くものは、ASCIZ 文字列のプログラムに、1 組のワード数を渡す引数（初期状態）です。もし、カレントディレクトリでファイルが見つかると、ASCIZ 文字列は EXEC システムコールと同じように実行可能なプログラムのドライブ名とパス名を渡します。もし、設定されたパスでファイルが見つかると、ファイル名はパスの情報とリンクされたものになります。プログラムは、この領域をプログラム自身がロードされた場所を知るのに使われます。コマンドプロセッサによって作成された環境（コールを行ったすべてのプログラムに渡された）には、少なくとも文字列〝COMSPEC＝〟が入っています（COMSPEC のパラメータは、ディスク上の COMMAND.COM の位置指定を行うために MS–DOS によって使用されるパスを定義します）。PATH と PROMPT コマンドも、MS–DOS の SET コマンドを通して入力されたすべての環境文字列と一緒に環境の中に入れられます。

ユーザープログラムに渡された環境の実際は、コールを行った環境のコピーです。プログラムを〝メモリに常駐させたまま終了〟させていると、プログラムに渡された環境のコピーが静的であることに注意しなければなりません。すなわち、次に SET、PATH または PROMPT コマンドが入力された場合でも、このコピーは変更されません。逆に、元のプロセス環境で、アプリケーションによるコピー

された環境のどんな変更もできません。たとえば、SETコマンドなどで設定された MS–DOSの環境変数を変えることはできません。

・オフセット 50H

PSP内のオフセット 50Hから3バイトに、MS–DOSファンクションディスパッチャのコールを行うためのコード（INT 21H、RETF）が入っています。したがって、指定したいファンクション番号をAHに入れ、割り込みタイプ 21Hをかけるのではなく、PSP＋50Hに対する longコールによって MS–DOSファンクションを行うことができます。これはコールであり、割り込みではないので、この位置にシステムコールを行うための該当するすべてのコードを入れることができます。これによって、システムのコールを行う処理を、移植性のあるものにすることができます。

・オフセット 80H

80Hには、コマンド行で指定されたパラメータの文字数が入っています。この次にパラメータの文字列が入ります（区切り記号も含めて）。ディスク転送アドレス（DTA）は、80Hにセットされます（PSP内のデフォルト DTA）。この DTAを使用した場合は、パラメータが破壊されます。

・オフセット 5CH、6CH

PSPのグラムセグメントの 5CHおよび 6CHのファイルコントロールブロック（FCB）には、コマンドが入力されたとき、先頭の2つの FCBがセットされます。いずれかのパラメータにパス名が入っている場合、対応する FCBには有効なドライブ番号のみが入っており、ファイル名フィールドは無効になります。

・オフセット 06H

オフセット 06H（1ワード）には、セグメント内の使用可能なバイト数が入っています。

・オフセット 02H

オフセット 02H（1ワード）には、利用できないメモリの先頭バイトを示すセグメントアドレスが入っています。プログラムは、ファンクションリクエスト 48H（メモリの割り当て）が行われるまで、このアドレスを変更してはいけません。

AXレジスタには、先頭の2つのパラメータ中のドライブ名が妥当かどうかを表す情報が返されます。

AL＝FFH　第1のパラメータに、無効なドライブ名が入っている場合
　　　　　（他は、AL＝00H）
AH＝FFH　第2のパラメータに、無効なドライブ名が入っている場合
　　　　　（他は、AH＝00H）

EXE 形式と COM 形式のプログラムでは、起動時に次のレジスタがセットされます。

・EXE 形式

DS、ES レジスタは、PSP の先頭を示すようにセットされます。CS、IP、SS、SP レジスタは、リンカによって渡された値にセットされます。

・COM 形式

4 つのセグメントレジスタは、PSP の先頭を示すようにセットされます。

すべてのユーザーメモリが、プログラムに割り当てられます。あるプログラムから EXEC ファンクションコールによって他のプログラムのコールを行う場合には、最初にセットブロック（ファンクション 4A00H）ファンクションコールでいくらかのメモリを解放し、第 2 のプログラムのためのスペースを用意しなければなりません。

命令ポインタ（IP）は、100H にセットされます。

SP レジスタは、プログラムセグメントの終わりにセットされます。オフセット 06H にあるセグメント内の使用可能なメモリのバイト数は 100H だけ縮小され、この大きさのスタックが使用可能になります。

スタックのトップには 0000H(1 ワード)がPUSHされます。これはユーザープログラムが、RET によって COMMAND に戻るためのものです。ただしそのために、ユーザープログラムがスタックとコードセグメントを管理することを前提としています。

第5章

プログラムヒント

5.1 イントロダクション

本章では、将来のMS–DOSのバージョンに対応するための、バージョン3.3でのプログラム手順について解説します。

5.2 割り込みタイプ

・割り込みタイプ22H

割り込みタイプ22H（終了アドレス）は、絶対にユーザーが実行してはいけません。この割り込みタイプは、MS–DOS自身だけが実行できます。

・割り込みタイプ23Hと24H

割り込みタイプ23H（<CTRL–C>の抜け出しアドレス）は、絶対にユーザーが実行してはいけません。この割り込みタイプは、MS–DOS自身だけが実行できます。

・割り込みタイプ24H

割り込みタイプ24H（致命的エラーによる中断アドレス）は、注意して使用してください。割り込みタイプ24Hハンドラは、システムコールの01H〜0CH、30H、59Hについてのみ実行できます。これ以外のコールを行うと、スタックが破壊され、「再試行する」または「無視する」を選択した場合の処理が正しく行われなくなります。

割り込みタイプ24Hハンドラは、ESレジスタを保存しなければなりません。また、プログラムを「再試行する」か、「無視する」を選択したとき、レジスタSS、SP、DS、BX、CX、DXを保存してください。

割り込みタイプ24Hは、選択の回答を受け取ると、回答を伴いIRETによってMS–DOSに戻ります。

割り込みタイプ24HでIRETを実行しないプログラムでは、01Hから0CH以外のコールをするまで、システムが不安定な状態となります。「無視する」を選択すると、不正なデータや無効なデータが内部システムバッファに残ります。

割り込みタイプ23H（<CTRL–C>の抜け出しアドレス）と割り込みタイプ24H（致命的エラーによる中断アドレス）のトラップは避けてください。特に割り込みタイプ24Hによるトラップエラーを、コ

ピー保護などの目的で使ってはいけません。この方法は、将来のMS–DOSのバージョンで使用できなくなる可能性があります。

・割り込みタイプ25Hと26H

プログラムが割り込みタイプ25H（アブソリュートディスクリード）、または26H（アブソリュートディスクライト）を実行する前に、すべてのレジスタをセーブしてください。また、プログラムの互換性や信頼性という点で問題があるので、これらの割り込みの使用はできるだけ避け、通常のファイル操作をもちいてディスクにアクセスしてください。

これらの割り込みは、セグメントレジスタを除くすべてのレジスタを破壊します。

・割り込みベクタの読み書き

メモリに、またはメモリから割り込みベクタを直接、書き込みまたは読み出しすることは避けてください。

ファンクション25H（割り込みベクタをセットする）と35H（割り込みベクタを得る）によって、割り込みテーブル中の値を得る、またはセットすることができるので、システムコールから割り込みベクタを操作してください。

5.3　システムコール（ファンクションリクエスト）

プログラムがMS–DOSバージョン2.0以前と互換性を保つ必要のある場合を除いて、システムコールは新しい方を使ってください。詳細については、1.8「バージョン2.0以前のシステムコール」を参照してください。

ファンクション01Hから0CHと26H（新しいPSPを作成する）を使うことは避けてください。標準入出力の読み出し、書き込みには、新しいシステムコールを使用してください。子プロセスを起動するときは、ファンクション26Hの代わりにファンクション4B00H（プログラムのロードと実行）を使います。

複数の処理を行っているときは、ファイルシェアリングのシステムコールを使います。詳細については、1.5「ファイルとディレクトリの管理」を参照してください。

MS–Networksには、ネットワークのシステムコールを使います。IOCTLの様式のいくつかは、MS–Networks用に用意されたものです。詳細については、1.6「MS–Networks」を参照してください。

ファンクション0EH（ディスクの選択）によってディスクを選択するときは、ALに返された値を注意して扱ってください。ALの値は、論理ドライブの最大数を返しますが、どのドライブが有効であるかは示していません。

5.4　デバイス管理

インストール可能なデバイスドライバ（装置ドライバ）を使ってください。MS–DOSは、BIOSを追

加できる構造をもつため、CONFIG.SYS にデバイスドライバを登録することによって、ブート時にドライバをインストールすることができます。転送するデータの単位は、ブロックデバイスドライバが一度に1ブロック、キャラクタデバイスドライバは1バイトです。

この2つのデバイスドライバについての詳細は、プログラマーズリファレンスマニュアル Vol.2「MS–DOS デバイスドライバ」を参照してください。

デバイスドライバは、バッファリングされた I/O を使います。データストリームは 64K バイトまでバッファリングすることができます。

大量のデータを画面に出力する場合、1回のコールで行うことができ、効率が上がります。

ファンクション 06H と 07H（直接コンソール I/O と直接コンソール入力）を使用し、直接コンソールと I/O を行うプログラム、<CTRL–C>をデータとして読み取るプログラムは、〝<CTRL–C>の検査〟がオフになっていることを確認する必要があります。

プログラムでは、この〝<CTRL–C>のチェック〟がオフになっているかどうかは、ファンクション 33H を使って確認できます。

5.5　メモリ管理

MS–DOS は、各メモリ領域の先頭にメモリコントロールブロックを置くことによって割り付けられたメモリを管理します。プログラムはファンクション 48H（メモリの割り当て）、49H（割り当てられたメモリの解放）、4AH（割り当てられたメモリブロックの変更）を使って、不要なメモリを解放します。

これらの処置は、将来のバージョンに対し、互換性を保つために有効です。

メモリ管理の詳細については、1.3「メモリ管理」を参照してください。

システムコールのメモリ管理によって得られた領域以外のメモリを、直接アクセスしてはいけません。絶対アドレス指定ではなく、相対アドレス指定のみを使用します。

プログラムが割り当てられていないメモリ領域を使用した場合、他のアプリケーションプログラムを破壊したり、最悪の場合には、MS–DOS のロードされている領域を破壊してシステムをダウンさせることもあります。

5.6　プロセス管理

EXEC ファンクションコールによって、プログラムのロードと実行を行います。プログラムのロードおよびオーバーレイには、EXEC ファンクション（ファンクション 4BH）を使います。EXE 形式のファイルのヘッダを参照して、直接ロードすることは避けてください。EXEC ファンクションコールを使うことによって、将来の MS–DOS のバージョンで EXE 形式のファイルのフォーマットが変更されても、互換性が保証されます。

割り込みタイプ 27H（プログラムをメモリにとどめたまま終了）の代わりにファンクション 31H（キー

ブプロセス）を使います。ファンクション31Hは64Kより大きいプログラムにも対応しています。

プログラムの終了には、ファンクション4CH（プロセスの終了）を使います。次の手順のいずれかによって、終了するプログラムはCSレジスタがプログラムセグメントプレフィクス（PSP）のセグメントアドレスを含んでいることを確認しなくてはなりません。

PSP内のオフセット0にロングジャンプを行う
INT 20Hを行う（CS：0はPSPを指していること）
AH = 0で、INT 21Hを行う（CS：0はPSPを指していること）
AH = 0で、PSPのロケーション50Hに対するロングコール

5.7 ファイルとディレクトリの管理

MS–DOSのファイル管理システムを使います。MS–DOSファイルシステムを使うことによって、将来のMS–DOSのバージョンに対し、ディスクフォーマットとディスク内のファイル／ディレクトリ管理の方法という点から、互換性が保証されます。

FCBの代わりに、ファイルハンドルを使います。ハンドルとは、ファンクション3CH（ハンドルの作成）、3DH（ハンドルのオープン）、5AH（一時ファイルの作成）、5BH（新しいファイルの作成）によって、ファイルがオープンまたは新規作成されるとき、MS–DOSが返す16ビットの値で、MS–DOSは以後このハンドルを通じてファイルにアクセスします。ハンドルを使用するMS–DOSのファイル管理のファンクションリクエストは、1.5「ファイルとディレクトリの管理」を参照してください。

このコールは、FCB（ファイルコントロールブロック）を使うバージョン2.0以前のファイル管理のファンクションの代わりに使われますこれは、ファイル操作が、FCB情報を操作せず、単にそのハンドルを操作するためです。

FCBを使わなければならないときは、プログラムがFCBを保護して、内容が書き換えられたりしないように気をつけなければなりません。

割り込みタイプ20H（プログラムの終了）、ファンクション00H（プログラムの終了）、ファンクション4CH（プロセスの終了）、ファンクション0DH（リセットディスク）を実行する前に、長さを変更したファイルをすべてクローズします。

変更したファイルがクローズされていないと、ファイルの長さが、正しく書き込まれません。

必要のなくなったファイルは、すべてクローズします。これによって、ネットワーク環境の状況が最適化されます。

ディスク上のすべてのファイルがクローズされていないかぎり、ディスクを変更してはいけません。内部システムバッファ上の情報が変更を受けた場合、ディスク上に不正確に書き込まれることがあります。

ロックファイル

プログラムは、ロックされている領域へアクセスが禁止されているかどうかは認識できません。ファイルのロック(ファンクション5C00H)を試行し、エラーコードを調べることによって、領域の状態を確認してください。

プログラムは、ロックされた領域を含むファイルをクローズしたり、ロックされた領域を含むオープンしたファイルをそのままにして終了することは許されません。この場合、結果は保証されません。割り込みタイプ 23H（<CTRL–C>の抜け出しアドレス）、または割り込みタイプ 24H（致命的エラーによる中断アドレス）によって終了する可能性のあるプログラムは、この割り込みをトラップし、抜け出す前にすべてのロックされた領域を解放しなければなりません。

5.8　その他のプログラム手順

タイミングに対する依存

CPU のクロックや処理速度は機種によって異なります。CPU の速度やクロックのタイミングに依存するプログラムは、旧機種や新機種では正常に動作しないことがあるので注意してください。

また、ネットワーク環境内では、タイミングにクロックを使用するプログラムは、信頼性が低下します。

指定された MS–DOS インターフェイスの使用

ハードウェア、またはメディアが変更されても、MS–DOS の提供するインターフェイスを使用していれば、プログラムの変更なしに、それらの機能を使うことができます。

ですから、MS–DOS でサポートしていないファンクションコール、割り込み、機能を設定したり使用したりしないでください。将来の MS–DOS のバージョンで、変更がなされ、同一名で定義されるかもしれないからです。そのようにして作成されたプログラムは極めて互換性の低いものとなります。

VRAM の直接アドレス指定不可

グラフィックを扱う際には、グラフィック用デバイスドライバを利用するようにしてください。VRAM を直接アクセスするのはできるだけ避けてください。VRAM のアドレスは機種によって異なるため、異機種間の互換性がまったく失われるからです。グラフィックドライバの詳細については、MS–DOS プログラマーズリファレンスマニュアル Vol.2 の「デバイスドライバ」を参照してください。

COM 形式より EXE 形式

EXE 形式のファイルは、リロケータブル（再配置可能）ですが、COM 形式のファイルはメモリイメージをそのままもつファイルです。COM 形式のファイルには、リロケーションのためのコントロール情報が含まれていないため、リロケーションは行われません。EXE 形式のファイルは、将来の MS–DOS のバージョンと互換性を保つための拡張可能なヘッダをもっています。

環境を使った情報の受け渡し

親プロセスの環境（SET コマンド等で設定された環境変数）は、子プロセスに引き継がれます。COMMAND.COM は、通常、すべてのアプリケーションの親プロセスになるので、カレントドライブとパス情報を、容易にアプリケーションに渡すことができます。

付録A

EXEファイルの構造とローディング

リンカユーティリティ（LINK）によって生成されたEXE形式のファイルは、次の2つの部分によって構成されています。

1. リロケーション及びコントロール情報
2. ロードモジュール

コントロール情報、リロケート（再配置）情報は、ファイルの先頭の〝ヘッダ〟と呼ぶ領域に入っています。ロードモジュールはヘッダのすぐ後に位置します。ロードモジュールは、パラグラフの境界から開始し、リンカによって生成されるモジュールのメモリイメージのことです。

ヘッダは、次のようなフォーマットをしています。

オフセット（16進）	内　容
00〜01H	4DH、5AH。ファイルが有効なEXE形式のファイルであることを示すためLINKプログラムによって付けられたマーク
02〜03H	最後のページに入っているバイト数、オーバーレイによる読み込みに有用
04〜05H	512バイト（ページ）単位の、ファイルの大きさ（ヘッダも含む）
06〜07H	リロケーションテーブルの項目数。この表はヘッダの直後に置かれる
08〜09H	ヘッダのサイズ（16バイトパラグラフ単位）ロードモジュールの開始点の位置指定に使用される
0A〜0BH	ロードされたプログラムの後に必要とされる16バイトパラグラフの最小数（minalloc）
0C〜0DH	ロードされたプログラムの後に必要とされる16バイトパラグラフの最大数（maxalloc）。minallocとmaxallocが両方ともゼロのときは、プログラムはできるだけ上位にロードされる
0E〜0FH	ロードモジュール内のスタックセグメントのオフセット（セグメントのフォーム）
10〜11H	モジュールに制御が渡されたとき、SPレジスタに返される値
12〜13H	ワードチェックサム−−オーバフローを無視した、ファイル内の全ワードのネガティブサム

オフセット（16進）	内　容
14～15H	モジュールに制御が渡されたとき、IPレジスタに返される値
16～17H	ロードモジュール内のコードセグメントのオフセット（セグメントのフォーム）
18～19H	ファイル内の先頭のリロケーション項目のオフセット値
1A～1BH	オーバーレイ番号（プログラムの常駐部分は0）

上記の項目の後に、リロケーションテーブルが置かれます。このテーブルは、変数のリロケーション項目によって構成されています。項目数は、オフセット06～07に入っています。リロケーション項目には2つのフィールド、2バイトのオフセット値と2バイトのセグメント値が入っています。これらの2つのフィールドには、モジュールに制御が渡される前に修正を必要とする、ワードのロードモジュール中のオフセットが入っています。このプロセスは、リロケーション（再配置）と呼ばれ、次のように処理されます。

1. ヘッダのフォーマットが行われている部分がメモリ中に読み込まれます。ヘッダの大きさは、1BHです。

2. メモリの一部がロードモジュールサイズとアロケーションユニット数（0A～0B、0C～0D）によってアロケートされます。MS–DOSは、パラグラフFFFFHのアロケートを試みます。これは常にエラーとなりますが、結果として、最大フリーブロック数が返されます。もし、このブロック数がminalloc及びロードサイズよりも小さいと、ノーメモリエラーとなります。また、もしこのブロック数がmaxallocとロードサイズよりも大きいと、MS–DOSはアロケートを行います（maxalloc＋ロードサイズ）。さもなければ、MS–DOSはメモリの最大フリーブロックにアロケートを行います。

3. PSPがアロケートされたメモリの最低位に作られます。

4. ロードモジュールの大きさは、ファイルの大きさ（オフセット04H～05H）からヘッダの大きさ（08H～09H）を引くことによって決定されます。実際の大きさはオフセット02～03の内容に基づき、調整が行われます。LINKのhigh/lowスイッチのセッティングによって、ロードモジュールをロードするための該当するセグメントが決定されます。このセグメントは、スタート（開始）セグメントと呼ばれます。

5. ロードモジュールが、スタートセグメントからメモリへロードされます。

6. リロケーションテーブル項目は、ワークエリアに読み込まれます。

7. 各リロケーションテーブル項目のセグメント値が、スタートセグメント値に加算されます。この計算されたセグメントは、リロケーション項目オフセット値とともに、ロードモジュール内のワードを示します。演算結果は、ロードモジュール内のワードに返されます。

8. いったんすべてのリロケーション項目が処理されると、SS、SP レジスタは、ヘッダ内の値によってセットされ、スタートセグメント値が SS に加算されます。ES、DS レジスタは、PSP 内のセグメントアドレスにセットされます。スタートセグメント値がヘッダ CS レジスタの値に加算され、この演算結果はヘッダ IP 値とともに、CS:IP の初期値としてこのモジュールに制御を渡すために使用されます。

付録 B

インテルオブジェクトモジュールフォーマット

B.1　イントロダクション

本章は、8086 マイクロプロセッサ（8086 および上位互換性のあるファミリーすべてを含む：以降、単に 8086 と呼びます）のリロケータブル（再配置可能）なオブジェクト言語を定義する、オブジェクトレコードのフォーマットについて解説します。8086 オブジェクト言語は、8086 をターゲットプロセッサとし、LINK でリンク（連結）可能な、すべての言語トランスレータの出力を指します（本章の解説では、アセンブラ、コンパイラを総称し、トランスレータと呼びます）。8086 オブジェクト言語はリンカやライブラリマネージャなどのオブジェクト言語プロセッサの入力であると同時に出力でもあります。

8086 オブジェクトモジュールのフォーマットを使うと、相互に連結可能でリロケータブル（再配置可能）なメモリイメージを指定することができます。このフォーマットは、8086 マイクロプロセッサのメモリマップ機能を、有効に使用できるように設計されています。

次の表は、マイクロソフト社が採用しているレコードフォーマットの一覧です。このレコードフォーマットは、本章で説明します。表中で、前にアスタリスク（*）マークがつけられたレコードフォーマットは、インテル社の仕様に準じたものであることを示します。

T–モジュール　ヘッダレコード
ネームリストレコード
*セグメント定義レコード
*グループ定義レコード
*タイプ定義レコード
シンボル定義レコード
*パブリック名定義レコード
*エクスターナル名定義レコード
*行番号レコード
データレコード
論理データレコード（繰り返し参照されない）
論理データレコード（繰り返し参照される）
FIXUP　レコード
*モジュールエンドレコード
コメントレコード

B.2　用語の定義

以下に、8086 の再配置とリンクの基礎となる用語を示します。

OMF

オブジェクトモジュールフォーマット（Object Module Format）。

MAS

メモリアドレス空間（Memory Address Space）。
8086MAS は 1M（メガ：1048576）バイトです。
この MAS は、実メモリ（MAS の一部分になる）と区別されることに注意してください。

MODULE（モジュール）

トランスレータによって生成したオブジェクトコードと、他の情報の分割不可能な集合。

T–MODULE（T–モジュール）

PASCAL や FORTRAN のようにコンパイラ／アセンブラが生成したモジュール。

オブジェクトモジュールは、次の制限を受けます。

1. 各モジュールには名前がつけられます。トランスレータは、T–モジュールに名前を与えますが、ソースコードも使用者も他の名前を指定しないと、デフォルト名（通常ファイル名、または空名称）が使われます。

2. シンボリックデバッガが各種の行番号やローカルなシンボルを読み分けることができるように、リンクされたモジュールの集合体の中の各 T–モジュールは、それぞれ別の名前がつけられます。このような制限はリンカが要求するものではなく、また強制するものでもありません。

FRAME（フレーム）

パラグラフ境界（16 の整数倍のアドレス）で始まる、MAS で 64K の連続域。8086 の 4 つのセグメントレジスタの内容が 4 つの（重なりあっても良い）フレームを定義するため、このような 8086 コードの 16 ビットアドレスでは、その時点での 4 つのフレーム以外のメモリロケーションをアクセスすることはできません。

LSEG（論理セグメント）

コンパイル／アセンブル時（アドレス拘束時以外）に内容を決定されるメモリの連続域。MAS 中のサイズおよびロケーションをコンパイルするとき、決定する必要はありません。リンク時に LSEG は、他の LSEG と結合して 1 つの LSEG を形成するため、サイズは、各 LSEG 内で部分的に固定されますが、最終的なものでありません。LSEG は、フレーム内に収まらなければならないので、サイズは 64K バイト以内です。LSEG のどの領域も、その LSEG を含むフレームのベースから、16 ビットオフセットだけでアドレス指定することができます。

PSEG（物理セグメント）

この語はフレームと同一です。「PSEG」と「LSEG」は、セグメントが「物理的」か「論理的」かの区別を表しているので、場合によって、この語が選んで使われることもあります。

FRAME NUMBER（フレーム番号）

各フレームはパラグラフ境界から始まります。MASの「パラグラフ」は0から65535までの番号を付けることができます。この番号は、それぞれフレームを定義するため、フレーム番号と呼ばれます。

PARAGRAPH NUMBER（パラグラフ番号）

フレーム番号と同一です。

PSEG NUMBER（PSEG番号）

フレーム番号と同一です。

GROUP（グループ）

コンパイルまたはアセンブル中に決まるLSEGの集合のこと。その集合のMAS中における最終的な位置は、その集合中の各LSEGをカバーできるフレームが少なくとも1つは存在しなくてはならないという制約を受けています。

「Gr A(X、Y、Z)」は、LSEGでX、Y、ZがAという名前のグループを形成することを示しています。X、Y、Zが同じグループに含まれるLSEGであっても、MAS中のX、Y、Zの順番や、X、Y、Z間の連続性を表すものではありません。

現在、マイクロソフトLINKでは、LSEGを複数のグループに属させることはできません。リンカは、複数のグループへのLSEGの位置づけを無視します。

CANONIC（正規）

MAS中のロケーション（アドレス）に注目して見ると、それを含むフレームは4096通り考えることができます。

この4096通りのフレームの中のフレーム番号の最大のものだけを区別して、特別にそのロケーションの正規フレームと呼びます（あるバイトの正規フレームとは、そのフレームからのバイトオフセットが0〜15の範囲に入るように選択されたフレームということができます）。したがってFOOがメモリロケーションを定義したシンボルであると、「FOOの正規のフレーム」というように使うことができます。

拡張すると（Sを何かメモリロケーションの集合としたとき）、Sの中のロケーションでの正規フレームの集合中で、最下位のフレーム番号をもつフレームはただ1つ存在します。この特定のフレームを、集合Sの正規フレームと呼びます。したがって、LSEGの正規フレームやLSEGのグループの正規フレームとか呼ぶことができます。

SEGMENT NAME（セグメント名）

LSEGはコンパイルまたはアセンブル時に、セグメント名を割り当てられます。この名前の割り当ては、次の目的で行われます。

1. リンク時にどのLSEGが他のLSEGと連結されるのかを決める役割を果たします。
2. グループを指定するために、アセンブラリソースコード中で使用されます。

CLASS NAME（クラス名）

LSEG には、翻訳時に、オプションでクラス名を割り当てることができます。同じクラス名をもつ 2 つの LSEG は、同一クラスに属していることになります。

LINK は、次の意味で名前をクラス付けします。「CODE」というクラス名や、語尾に「CODE」を含むクラス名は、そのクラスがコードのみを含んでおり、読み出すことしかできないことを意味します。このようなセグメントのとき、オーバーレイの一部として、そのセグメントを含むモジュールを指定すると、オーバーレイすることができます。

OVERLAY NAME（オーバーレイ名）

LSEG には、オプションとしてオーバーレイ名を割り当てることができます。LINK（バージョン 2.40 以降）は、LSEG オーバーレイ名を無視しますが、インテルの再配置（relocation）と連結（リンク：linkage）のツールでは、これを使用することができます。

COMPLETE NAME（コンプリート名）

LSEG のコンプリート名は、セグメント名、クラス名、オーバーレイ名で構成されます。別々のモジュール中の LSEG は、そのコンプリート名が同一であれば、リンク（連結）されます。

B.3　モジュールの一致と属性

モジュールのヘッダレコードは、モジュール中で常に最初のレコードとなり、これがモジュール名を与えます。

名前を与えられたモジュールは、指定された開始アドレスをもつものと同様に、主プログラムとしての属性をもつことができます。複数のモジュールを連結するときには、主プログラムの属性をもつモジュールを 1 つだけ与えます。

これにはモジュールが主プログラムになる場合とならない場合があり、また開始アドレスをもつ場合ともたない場合があることを示します。

B.4　セグメント定義

モジュールは、トランスレータによって生成される、レコードの並びによって定義されているオブジェクトコードの集まりです。オブジェクトコードは、コンパイルまたはアセンブル時に内容を決定されるメモリの連続域を表しています。この領域を論理セグメント（LSEG）と呼びます。

モジュールは、各 LSEG の属性を定義します。セグメント定義レコード（SEGDEF）は、すべての LSEG 情報（名前、レコード長、メモリ配置等）を維持する媒体です。複数の LSEG がリンクされていて、セグメントアドレス可能性（A.5「セグメントアドレッシング」を参照してください）が確立されているとき、LSEG 情報が必要になります。SEGDEF レコードは、最初のヘッダレコードの後に置かれなければなりません。

B.5　セグメントアドレッシング

8086には、64Kバイトのメモリ領域（フレームと呼ばれる）をアドレッシングするためにセグメントベースレジスタが用意されています。これらには1つのコードセグメントベースレジスタ（CS）と2つのデータセグメントベースレジスタ（DS、ES）、1つのスタックセグメントベースレジスタ（SS）があります。

メモリイメージを作り上げるLSEGの数の最大値は、使用可能なベースレジスタの数をはるかに上まわります。したがって、ベースレジスタは、そのたびにロードする必要があります。たとえば、小さなデータLSEGやレコードLSEGが、たくさん集まって作られたモジュールプログラムなどがそれに当たります。

ベースレジスタを、そのたびにロードするのはあまり望ましくないため、1つのメモリフレームに納まる単一ユニットに、多くの小さいLSEGを集め、同じベースレジスタ値を使用して、すべてのLSEGをアドレッシングできるようにするのがよいでしょう。このアドレッシング可能なユニットはグループといい、A.2「用語の定義」で定義されています。

グループ中のオブジェクトをアドレッシングできるようにするには、グループがモジュールの中で明確に定義されていなくてはなりません。グループ定義レコード（GRPDEF）は、セグメント名や、「シンボルFOOを定義するセグメント」または「ROMというクラス名をもつセグメント」のような属性などによって、構成セグメントのリストを与える必要があります。

モジュール中のGRPDEFレコードは、すべてのSEGDEFレコードの後に置かれなくてはなりませんが、これはグループを定義するために、GRPDEFレコードがSEGDEFレコードを参照するからです。また、GRPDEFレコードは、リンカが最初に処理しなくてはならないため、他のすべてのレコード（ヘッダレコードを除く）より先に置かれなければなりません。

B.6　シンボル定義

マイクロソフトLINKは、シンボル定義レコードのクラスになる3種類のレコードを採用しています。その中の、パブリック名定義レコード（PUBDEF）とエクスターナル名定義レコード（EXTDEF）の2つはいずれも重要です。これらはグローバルに参照可能なプロシージャとデータ項目を定義し、外部参照を解決するために使われます。さらにTYPDEFレコードは、マイクロソフトLINKが共有変数の割り当てをするために使われます。A.14「共有変数の型に関するマイクロソフト表現法」を参照してください。

B.7　インデックス

インデックスは、数値の項目の集合中から特定のものを選択する整数です。たとえば、名前インデックス、グループインデックス、エクスターナルインデックス、型インデックスなどがあります。

注意　インデックスは通常、正の数です。インデックス値の0は予約されており、インデックスの型によって特別な意味をじたせることもあります（つまり、セグメントインデックスが0のときは「名前なし」の擬セグメントであることを示し、また型インデックスが0のときは、「型なし」のセグメントで、「指定なし」とは区別されることを示すなどです）。
一般的に、インデックスは値が非常に大きいところまでを想定しています（つまり、255をはるかに超える）。しかしながら、オブジェクトファイルの多くは、50や100を超えるインデックスを含みません。したがって、必要に応じて、インデックスは1〜2バイトでコード化されます。
第1バイト（おそらくはこれのみ）の高位（最も左の）ビットは、インデックスが1バイトを占めるか2バイトを占めるかを決定します。そのビットが0であると、インデックスが0〜127になり、1バイトを占めます。そのビットが1であると、インデックスは0〜32767の値をとり、2つのバイトは下位8ビットが第2バイト、上位7ビットが第1バイトとなります。

B.8　フィックスアップのためのフレームの概念

「フィックスアップ」は、オブジェクトコードに与えるある変更で、これはトランスレータによって要求され、リンカによって実行し、アドレスの結合をします。

注意　前述の「フィックスアップ」の定義は、正確にはリンカの側からの視点を表します。しかしながら、リンカはこの定義に合わないオブジェクトコードの変更（すなわち、「フィックスアップ」）を行うために使われることもあります。たとえば（オブジェクト）コードのハードウェア浮動小数点、またはソフトウェア浮動小数点サブルーチンへの連結は、オペレーションコードの変更になります（このときオペレーションコードはアドレスとして取り扱われている必要があります）。前出の「フィックスアップ」の定義は、オブジェクトコードの変更を禁じるものでも軽んじるものでもありません。

8086のトランスレータは、次の4つのデータを与えることによって、フィックスアップを指定します。

1. フィックスアップするロケーションの場所と型
2. 2つあるフィックスアップ。MODE（モード）のうちのどちらか
3. ターゲット。ロケーションが参照しなくてはならないメモリアドレス
4. 参照した文脈を定義するフレーム

LOCATION（ロケーション）

ロケーションは5種類ありますが、それはポインタ、ベース、オフセット、HIBYTE（高位バイト）、LOWBYTE（低位バイト）です。

次の図の縦のアライメントは、4つの点を示します（8086メモリの1ワード中の高位バイトとは高位のアドレスをもつバイトであることに注意してください）。

1. ベースはポインタ中の高位ワードです（リンカはポインタの低位ワードが存在するか否かには関与しません）。

2. オフセットはポインタの低位ワードです（また、リンカは高位ワードが続くか否かには関与しません）。

3. HIBYTEはオフセットの高位側の半分です（リンカは、低位側の半分が前にあったか否かには関与しません）。

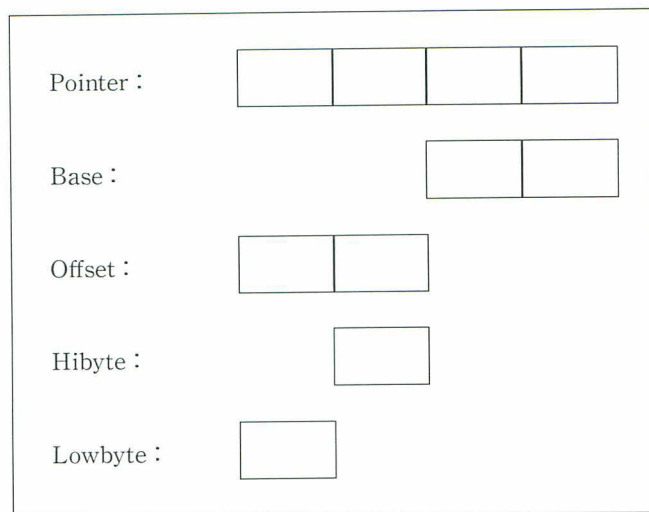
4. LOBYTEはオフセットの低位側の半分です（リンカは高位側の半分が存在するか否かには関与しません）。

Pointer：

Base：

Offset：

Hibyte：

Lowbyte：

ロケーションは、2つのデータによって指定されます。(1) ロケーションの型と (2) ロケーションの場所です。(1) はフィックスアップレコード中のLOCATフィールドのLOCサブフィールドによって指定します。(2) はフィックスアップレコード中のLOCATフィールドのDATA RECORD OFFSETサブフィールドで指定します。

MODE

リンカは2種類のフィックスアップである「セルフリラティブ（自己相対）」と「セグメントリラティブ（セグメント相対）」をサポートします。

自己相対フィックスアップは、CALL、JUMP、SHORT–JUMP 命令に使う 8 ビットと 16 ビットオフセットをサポートします。セグメント相対フィックスアップは、他のすべての 8086 アドレッシングモードをサポートします。

TARGET

ターゲットは MAS 中の参照されるロケーションです（正確には、ターゲットは参照されるオブジェクトの最下位バイトです）。ターゲットは、次の 8 つの方法のうちの 1 つで指定されます。そのうち 4 つは「基本的」な方法であり、他の 4 つは「二次的」な方法です。ターゲットを指定する基本的な方法では、インデックス、またはフレーム番号 X と変位 D の 2 種類のデータを使用します。

T0　X はセグメントインデックス。ターゲットはインデックスによって識別される LSEG の D 番目のバイトです。

T1　X はグループインデックス。ターゲットはインデックスによって識別される LSEG の D 番目のバイトです。

T2　X はエクスターナルインデックス。ターゲットは、インデックスによって識別されるエクスターナル名によって（結果的に）アドレスが与えるバイトの後の D 番目のバイトです。

T3　X はフレーム番号。フレーム番号によって識別されるフレーム中の D 番目のバイトです（つまりターゲットのアドレスは (X*16)+D のようになります）。

ターゲットを指定する「2 次的」方法は、どちらもデータ項目を 1 つだけとります。それは、インデックス、またはフレーム番号（インデックス、またはフレーム番号 X）です。変位は 0 であると仮定します。

T4　X はセグメントインデックス。ターゲットはインデックスにより識別される LSEG の 0 番目（最初の）のバイトです。

T5　X はグループインデックス。ターゲットは、MAS 中で結果的に最下位に位置づけされる指定グループ中の LSEG の 0 番目（最初の）バイトです。

T6　X がエクスターナルインデックス。ターゲットはインデックスによって識別されるエクスターナル名のアドレスとなるバイトです。

T7　X はフレーム番号。ターゲットは 20 ビットアドレスが (X*16) となるバイトです。

注意　LINK では前述のうち T3 と T7 の方法は使えません。

ターゲットを記述するとき、次のような表記法を使います。

TARGET：SI(〈セグメント名〉), 〈変位〉　[T0]
TARGET：GI(〈グループ名〉), 〈変位〉　[T1]
TARGET：EI(〈シンボル名〉), 〈変位〉　[T2]
TARGET：SI(〈セグメント名〉)　[T4]
TARGET：GI(〈グループ名〉)　[T5]
TARGET：EI(〈シンボル名〉)　[T6]

次に、これらの表記の例を示します。

TARGET：SI(CODE),1024
　セグメント「CODE」中の 1025 番目のバイト
TARGET：GI(DATAAREA)
　MAS 中の「DATAAREA」という名前のグループのロケーション
TARGET：EI(SIN)
　外部サブルーチン「SIN」のアドレス
TARGET：EI(PAYSCHEDULE),24
　「PAYSCHEDULE」という名称の外部データ構造の次に、24 番目のバイト

FRAME（フレーム）
　各 8086 メモリ参照は、いずれかのフレームに含まれるロケーションに向けられます。またフレームに、いずれかのセグメントレジスタの内容によって指定されます。リンカにとって正確で、かつ使用可能なメモリ参照を行うには、何がターゲットであり、参照すべきフレームがどこにあるかを与えなくてはなりません。このように、各フィックスアップはしかるべきフレームを、6 通りの方法のうちの 1 つによって指定します。方法によって、前述のように、インデックス、またはフレーム番号中のデータ X を使うものがあります。これ以外はデータを必要としません。

　次に、フレームを指定する 6 つの方法を示します。

F0　X はセグメントインデックス。フレームはインデックスによって定義される LSEG の正規フレームです。
F1　X はグループインデックス。フレームはグループによって定義される正規フレームです。(つまりグループ中で最終的に MAS 中で最下位に位置づけされた LSEG によって定義される正規フレーム)。
F2　X はエクスターナルインデックス。フレームはエクスターナル名のパブリック定義がなされると決定されます。これらは、次の 3 つに分けることができます。
F2a シンボルをある LSEG に相対的に定義し、相互に関連するグループがないとき。LSEG の正規フレームが指定されます。
F2b シンボルは LSEG を参照することなしに絶対的に定義され、相互に関連するグループがないとき。フレームは、シンボルを定義する PUBDEF フィールドのサブフィールドであるフレーム番号によって指定されます。
F2c シンボルの定義方法に無関係で、相互に関連するグループが存在するとき。グループの正規フレームによって指定されます。グループは、PUBDEF のサブフィールドであるグループインデックスによって指定されます。
F3　X はフレーム番号。これは明確にフレームを指定します。
F4　X がない場合、フレームはロケーションを含む LSEG の正規フレームです。
F5　X がない場合、フレームはターゲットによって決定されますが、次の 4 つに分けることができます。
F5a ターゲットがセグメントインデックスを指定するとき。この場合、フレームは (F0) と同様に決定されます。
F5b ターゲットがグループインデックスを指定するとき。この場合、フレームは (F1) と同様に決定されます。
F5c ターゲットがエクスターナルインデックスを指定するとき。この場合、フレームは (F2) と同様に決定されます。

F5d ターゲットが明示フレーム番号を指定するとき。この場合、フレームは (F3) と同様に決定されます。

注意　LINK ではフレーム指定法のうち F2b、F3、F5d は使えません。

フレームを記述するときも、ターゲットの記述と同様に行います。

```
FRAME：SI(〈セグメント名〉)    [F0]
FRAME：GI(〈グループ名〉)      [F1]
FRAME：EI(〈シンボル名〉)      [F2]
FRAME：LOCATION               [F4]
FRAME：TARGET                 [F5]
FRAME：NONE                   [F6]
```

8086 メモリ参照は、自己相対参照によって指定されるフレームが、通常ロケーションを含む LSEG の正規フレームであり、セグメント相対参照によって指定されるフレームはターゲットを含む LSEG の正規フレームです。

B.9　セルフリラティブフィックスアップ

セルフリラティブ（自己相対）フィックスアップは、次のように行われます。

メモリアドレスはロケーションによって暗黙の内に定義されます。つまり、ロケーションに続くバイトのアドレスにより定義されます（自己相対参照時に、8086 の IP（インストラクションポインタ）は、参照に続くバイトを指すためです）。

8086 の自己相対参照のとき、ロケーション、またはターゲットが指定フレームの外にあると、リンカは警告を出します。その他の場合、ロケーションが暗黙に定義するアドレスに加えられ、一義の 16 ビット変位が存在します。フレーム中のターゲットの相対位置を与えることになります。

ロケーションがオフセットであると、変位はロケーションに加えられ、65536 で割った余りが取られます。これは、エラーになりません。

ロケーションが LOBYTE であると、変位は−128〜127 の範囲でなければなりません。それ以外の場合は、リンカが警告を発します。変位はロケーションに加えられ、255 で割った余りが取られます。

ロケーションがベースポインタ、または HIBYTE であると、トランスレータ中で何が行われるのか、明確に表されてなく、リンカの行う動作も定義されていません。

B.10　セグメントリラティブフィックスアップ

セグメント相対フィックスアップは、次のように行われます。

負でない 16 ビット数 FBVAL は、フィックスアップが指定するフレームのフレーム番号として定義されます。さらに符号付き 20 ビット数 FOVAL は、フレームのベースからターゲットまでの距離として

定義されます。この符号付きの 20 ビット数が 0 より小さいか、または 65535 より大きいと、リンカはエラーを表示します。それ以外の場合、FBVAL、FOVAL は、次のようにロケーションをフィックスアップするのに使われます。

1. ロケーションがポインタであると、FBVAL は（MOD 65536：MOD は剰余計算）でポインタの高位ワードに加えられ、FOVAL は（MOD 65536 で）ポインタの低位ワードに加えられます。
2. ロケーションがベースの場合、FBVAL は（MOD 65536 で）BASE に加えられますが、FOVAL は無視されます。
3. ロケーションがオフセットである場合、FOVAL は（MOD 65536 で）オフセットに加えますが、FBVAL は無視されます。
4. ロケーションが HIBYTE の場合、（FOVAL/256）は（MOD 256 で）HIBYTE に加えられますが、FBVAL は無視されます（前述の除算は「整数除算」であり、余りは捨てられます）。
5. ロケーションが LOBYTE の場合、（FOVAL を 256 で割った余り）は（MOD 256 で）LOBYTE に加えられます。FBVAL は無視されます。

B.11 レコードオーダ

オブジェクトコードファイルは、1 個以上のモジュールの連続したものを含むか、0 以上のモジュールを含むライブラリを含む必要があります。1 つのモジュールは、オブジェクトコードの集合として定義され、コードはオブジェクトレコードの連続として定義されます。次の構文はモジュールを形成するための、レコードの正当な階層を示します。さらに与えられた構文規則は、レコード列の解決の方法に関する情報を与えます。

注意　次に使う構文記述言語は、WIRTH によって定義されています（CACM、1977 年 11 月作成、ボリューム#20、番号#11、#822–#832 ページ、大文字で書かれているのはリテラルではなく、レコードフォーマットの説明中で定義される識別子です）。

```
object file  =  tmodule
tmodule      =  THEADR seg-grp  {component}  modtail
seg_grp      =  {LNAMES}  {SEGDEF}  {TYPDEF | EXTDEF | GRPDEF}
component    =  data | debug_record
data         =  content_def | thread_def | TYPDEF | PUBDEF | EXTDEF

debug_record=  LINNUM
content_def  =  data_record  {FIXUPP}
thread_def   =  FIXUPP(containing only thread fields)
data_record    =  LIDATA   |   LEDATA
modtail        =  MODEND
```

次の規則が適用されます。

1. FIXUPPレコードは常に前のDATA（データ）レコードを参照します。
2. すべてのLNAME、SEGDEF、GRPDEF、TYPEDEP、EXTDEFのレコードは、これを参照するレコードより前に与えられていなくてはなりません。
3. COMENTレコードは、ファイル中のどこにも存在できますが、ファイルやモジュール中の最初、または最後のレコードとしたり、条件レコード中には置けません。

B.12　レコードフォーマットについて

次にレコードフォーマットダイアグラムの概略図を示します。これはレコードフォーマットのサンプルであり、各種の規則を表したものです。

■レコードフォーマットの例（SAMREC）

REC XYP xxH	RECORD LENGTH	NAME	NUMBER	CHK SUM
(1)	(2)	(1以上)	(4)	(1)
		RPT		

タイトルと公式略称

先頭には、図示したレコードフォーマットの名前と、その公式な略称が記述されています。トランスレータおよびデバッガのような種々のプログラム間で一義性を促進するため、コードとドキュメンテーションの双方でこの略称を使うべきです。レコードフォーマットの略称は、常に6文字で示されます。

ボックス

フォーマットはボックスによって記述されます。() 内の数字は、そのフィールドのサイズ（バイト単位）です。

RECTYP（レコードの型）

各レコードの第1バイトは、0〜255の値を取り、レコードがどの型（RECORD type）であるかを示しています。

RECORD LENGTH（レコード長）

各レコードの第2フィールドは、レコードのバイト数（初めの2つのフィールドを除く）を含みます。

NAME（名前）

「NAME（名前）」と書かれたフィールドは、どれも次の内部構造をもちます。1バイト目はフィールド中の残りのバイト数を示します。残りのバイトは、バイトごとの文字列として翻訳（コンパイル／アセンブル）されます。

ほとんどのトランスレータは、ASCII 文字セットの部分集合であるように限定しています。

NUMBER（番号）

4 バイトの NUMBER フィールドは、符号なしの 32 ビット整数を示し、先頭の 8 ビット（最小有効桁）を第 1 バイト（最低位アドレス）に、続く 8 ビットを第 2 バイトに、という形で格納されています。

REPEATED OR CONDITIONAL FIELDS（反復または条件フィールド）

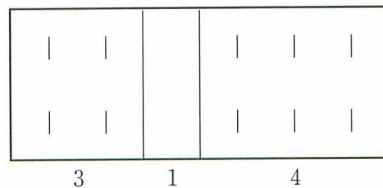
レコードフォーマットの一部には、数回反復されるフィールド列が含まれています。この部分は「RPT（反復）」というブラケットがボックスの下部に示されます。

同様に、与えられた条件が正であるか否かだけを示す部分もありますが、これは同じように「COND（条件）」というブラケットがボックスの下部に示されます。

CHKSUM（チェックサム）

各レコードの最後のフィールドはチェックサムです。これはレコード中の他のすべてのバイトの合計を、2 の補数（MOD 256）で表したものになっています。したがって、レコードに含まれるバイトの合計（MOD 256 で）は 0 になります。

BIT FIELDS（ビットフィールド）

フィールド内容の記述は、ビットレベルのこともあります。ボックス内に縦線（|）の引かれたボックスは、バイトまたはワードを示します。この縦線は、ビットの境界を意味し、次に示す図では、3 ビット、1 ビット、4 ビットの 3 つのビットフィールドがあることを示します。

■ T–モジュールヘッダレコード（THEADR）

REC TYP 80H	RECORD LENGTH	T-MODULE NAME	CHK SUM
(1)	(2)	(1 以上)	(1)

トランスレータから出力される各モジュールは、T–モジュールヘッダレコードをもちます。

T–MODULE NAME（T–モジュール名）

T–MODULE NAME は T–モジュールの名前です。

■ 名前リストレコード（LNAMES）

REC TYP 96H (1)	RECORD LENGTH (2)	NAME (1以上)	CHK SUM R (1)
		RPT	

このレコードは、続く SEGDEF や GRPDEF レコード中でセグメント名、クラス名や、またはグループ名として使われる名前のリストです。

モジュール中の LNAMES レコードの順序と、LNAMES レコード間での名前の順序は、名前の順序を付けることになります。したがって、それらの名前に 1、2、3、4、…と番号を割り当てることができます。この番号は、セグメント名やインデックス、クラス名インデックス、SEGDEF や GRPDEF レコードのグループ名インデックスフィールド中で「名前インデックス」として使われます。

NAME（名前）

この反復フィールドは、名前を示し、フィールド長が 0 をとることが可能です。

■ セグメント定義レコード（SEGDEF）

REC TYP 98H (1)	RECORD LENGTH (2)	SEG ATTR (1以上)	SEG MENT LEGNTH (2)	SEG MENT NAME INDEX (1以上)	CLASS NAME INDEX (1以上)	OVER LAY NAME INDEX (1以上)	CHK SUM (1)

特定の LSEG を参照するために、他のレコード型で使われるセグメントインデックス（セグメントインデックス）値（1～32767）は、オブジェクトファイル中に現れる SEGDEF レコード中で（列として）暗黙の内に定義されます。

SEG ATTR フィールドはセグメントの属性に関する情報を与え、次のフォーマットで示します。

ACBP (1)	FRAME NUMBER (2)	OFF SET (2)
	COND	

このACBPバイトは、属性を記述する4つの要素A、C、B、Pからなります。次に、このバイトのフォーマットを示します。

「A」（Alignment：アライメント（配置、配列））は、LSEGのアライメント属性を指定する3ビットのサブフィールドです。次にその意味を示します。

A＝0　SEGDEFは絶対LSEGを定義する
A＝1　SEGDEFはリロケータブルなバイトアライメントのLSEGを定義する
A＝2　SEGDEFはリロケータブルなワードアライメントのLSEGを定義する
A＝3　SEGDEFはリロケータブルなパラグラフアライメントのLSEGを定義する
A＝4　SEGDEFはリロケータブルなページアライメントのLSEGを定義する

A＝0の場合、フレーム番号フィールドとOFFSETフィールドが存在します。LINKでは、アドレス指定の目的のみに使用されます。たとえばROMの開始アドレスを定義し、ROM内にシンボル名を定義するなどです。LINKは、絶対LSEGに属するデータ指定を、すべて無視します。

「C」（Combination：結合タイプ）は、結合タイプを指定する3ビットのサブフィールドです。絶対セグメント（A＝0）は、結合タイプ0（C＝0）をもちます。リロケータブルなセグメントでのCフィールドは、セグメントがどのように組み合わせできるかを示す数（0、1、2、4、5、6、7）をコードとして使用します。この（結合タイプ）属性は、2つのLSEGの結合状態を考えると理解できるでしょう。X、YをLSEG、ZをX、Y結合タイプの結果（与えられるLSEG）と考えます。LX、LYをX、Yの長さ、MXYをLX、LYのうち大きい方とします。GはYのアライメント属性に適合するZ中のX、Y要素間のギャップとします。LZは（結合している）LSEG Zの長さ、dx（0≦dx <LX）はXのオフセット（バイト単位）、同様にdyは（バイト単位の）Yのオフセットとします。次の表は、結合しているLSEG Zの長さLZ、X中のdx、Y中のdyに対応する（Zに含まれる）オフセットであるdx'、dy'を表しています。インテルは、さらにアライメントタイプ5と6を定義し、そのアライメントタイプのセグメントのコードとデータを処理します。

C	LZ	dx'	dy'	
2	LX＋LY＋G	dx	dy＋LX＋G	"Public"
5	LX＋LY＋G	dx	dy＋LX＋G	"Stack"
6	MXY	dx	dy	"Common"

上記の表を見ると、C = 0、C = 1、C = 2、C = 4、C = 7に対応する行がありません。C = 0はリロケータブルなLSEGが結合されていない可能性があり、C = 1、C = 3は定義されません。C = 4、C = 7はC = 2と同様に取り扱われます。インテル規格では、C1、C4、C7が、すべて異なる意味をもちます。

「B」(Big) は1ビットサブフィールドで、これが1を取るとき、セグメント長がちょうど64K (65536) であることを示します。この場合、SEGMENT LENGTHフィールドは0でなければなりません。

「P」フィールドは常に0である必要があります。「P」フィールドは、インテル仕様である「ページ常駐」フィールドです。

フレーム番号とOFFSETフィールド（絶対セグメントA = 0のときのみ存在）は、絶対セグメントのMAS中の位置づけを指定します。OFFSETの範囲は0〜15に限られます。15以上の値をOFFSETに与えたいときは、フレーム番号を調整する必要があります。

SEGMENT LENGTH（セグメント長）

SEGMENT LENGTHフィールドは、セグメント長をバイト単位で与えます。長さは0でもかまいませんが、0であるとLINKはモジュールからセグメントを削除しません。セグメント長フィールドは、ちょうど0〜65535を格納できる大きさをもっています。セグメントにちょうど64Kの長さを指定するには、ACBPフィールドのB属性ビット（SEGATTRの項を参照してください）を使わなければなりません。

SEGMENT NAME INDEX（セグメント名インディックス）

セグメント名はプログラマー、またはトランスレータがセグメントに付ける名前です。たとえばCODE、DATA、TAXDATA、MODULENAME、CODE、STACKなどです。このフィールドは、LNAMEレコードが与える名称リストに、インデックス付けすることによってセグメント名になります。

CLASS INDEX（クラス名インディックス）

クラス名は、プログラマやトランスレータがセグメントに割り当てる名前です。割り当てられていない場合に、名前は空になり、長さは0になります。クラス名の目的は、MAS中のLSEGの順序づけに使うハンドルを（プログラマが）定義できるようにするためです。たとえばRED、WHITE、BLUE; ROM、FASTRAM、DISPLAYRAMなどです。このフィールドは、LNAMEレコードの与える名称リストに、インデックス付けすることによってクラス名を与えます。

OVERLAY NAME INDEX（オーバーレイ名インディックス）

注意　この項目は、バージョン2.40以降のLINKで無視されますが、それ以前のバージョンにはサポートされています。ただし、インテル仕様と意味

が異なります。

オーバーレイ名は、プログラマーの要求により、トランスレータまたはLINKがセグメントに付ける名前です。クラス名と同様、オーバーレイ名は空であってもかまいません。このフィールドは、LNAME レコードが与える名称リストにインデックス付けすることによりオーバレイ名を与えます。

注意　セグメントの「完全な名称」とは、セグメント名、クラス名、オーバレイ名３つの部分から成る名前です（後半の２つの名前は空とすることができます）。

■ グループ定義レコード（GRPDEF）

REC TYP 9AH (1)	RECORD LENGTH (2)	GROUP NAME INDEX (1以上)	GROUP COMPONENT DESCRIPTOR (1以上)	CHK SUM (1)
			REP	

GROUP NAME INDEX（グループ名インデックス）

グループ名は、LSEG が参照されるときに使う名前です。このグループの重要な特質として、結果的に LSEG が MAS 中で固定されるとき、グループの各 LSEG をカバーするフレームが存在しなくてはならないことがあげられます。

GROUP NAME INDEX フィールドは、LNAME レコードが与える名前のリストにインデックス付けすることによってグループ名を与えます。

GROUP COMPONENT DESCRIPTOR（グループ要素記述子）

次に、各 GROUP COMPONENT DESCRIPTOR のフォーマットを示します。

SI (FFH) (1)	SEGMENT INDEX (1以上)

記述子の第１バイトは 0FFH であり、前にある SEGDEF レコードが記述する LSEG を選択する SEGMENT INDEX フィールド１つを含みます。

インテルは、他にも４つのグループ記述タイプとそれぞれの意味を定義しています。これらは 0FFH、0FDH、0FBH、0FAH です。LINK は、これらすべてを 0FFH と同一として扱います（つまり、常に 0FFH にはセグメントインデックスが続くものとし、実際に値が 0FFH であるか否かをチェックしません）。

■ 型定義レコード（TYPDEF）

REC TYP 8EH (1)	RECORD LENGTH (2)	NAME (常に NULL) (1以上)	EIGHT LEAF DESCRIPTOR (1以上)	CHK SUM (1)
			└ REP ┘	

LINK は、TYPDEF レコードを共有変数の位置づけにのみ使用します。これは、インテルが目的としたものではありません。A.14「共有変数の型に関するマイクロソフト表現法」を参照してください。

必要な数の EIGHT LEAF DESCRIPTOR（8リーフ記述子）フィールドを使って、分岐を記述します（最後のレコードを除く）。この最後のレコードは、1〜8リーフを記述します。

可変の型インデックスの値（1〜32767）は、他のレコードタイプに（オブジェクトタイプとオブジェクト名を関連づけるために）含まれていますが、オブジェクトファイル中で、TYPDEF レコードを記述する順序によって暗黙の内に定義されます。

NAME（名前）

このフィールドの使用は予約されています。トランスレータは、このフィールドを0に（長さが0の名前の表現）しておきます。

EIGHT LEAF DESCRIPTOR（8リーフ記述子）

このフィールドは、8つまでのリーフを記述することができます。

E N (1)	LEAF DESCRIPTOR (1以上)
	└ RPT ┘

EN フィールドは1バイト、つまり8ビットで、（左から右の順に）8つのリーフが容易（ビット＝0）または精密（ビット＝1）であることを示します。

1〜8個の LEAF DESCRIPTOR（リーフ記述子）のフォーマットは、次のいずれかになります。

0～128
(1)

129～	0～64K − 1
(1)	(2)

132	0～16M − 1
(1)	(2)

136	− 2G + 1 ・ ・ ・ 2G − 1
(1)	(2)

第 1 のフォーマット（1 バイト）は、0～127 の値をもち、与えられた数値を値とする数字リーフを表現します。

第 2 のフォーマットは、先行バイトとして 129 で数字リーフを表現します。数値は続く 2 バイトに含まれます。

第 3 のフォーマットは、先行バイトとして 132 で数字リーフを表現します。数値は 3 バイトに含まれます。

第 4 のフォーマットは、先行バイトとして 136 があり、符号付き数字リーフを表現します。数値は、続く 4 バイトに含まれ、必要に応じて符号が付けられます。

■ パブリック名定義レコード（PUBDEF）

REC TYP 90H	RECORD LENGTH	PUBLIC BASE	PUBLIC NAME	PUBLIC OFFSET	TYPE INDEX	CHK SUM
(1)	(2)	(1 以上)	(1 以上)	(2)	(1 以上)	(1)
			RPT	RPT	RPT	

このレコードは、単一または複数の PUBLIC NAME のリストを与えますが、それぞれの名前ごとに 3 つのデータがあります。(1) 名前のベース、(2) 名前のオフセット値、(3) 名前の表現する実質の型の 3 つです。

PUBLIC BASE（名前のベース値）

PUBLIC BASEのフォーマットは次のとおりです。

GROUP INDEX (1以上)	SEGMENT INDEX (1以上)	FRAME NUMBER (2)
	COND	

GROUP INDEXフィールドのフォーマットは、すでに述べたように、0〜32767の値を取ります。0でないグループインデックスは、パブリックシンボルのついたグループに結び付き、A.8「フィックスアップのためのフレームの概念」のF2cの方法で使用されます。グループインデックスが0であると、関連グループがないことを示しています。

SEGMENT INDEXフィールドのフォーマットも、すでに説明したように、0〜32767の値を取ります。

0でないセグメントインデックスは、1つのLSEGを指定します。このとき、レコード中で定義される各パブリックシンボルのロケーションは、選択したLSEGの第1バイトからの負でない変位（PUBLIC OFFSETフィールドで指定します）として扱われ、フレーム番号は付けられません。

セグメントインデックス（グループインデックスが0のときのみ有効）が0であると、レコード中で定義されているパブリックシンボルのロケーションは、フレーム番号フィールドの値が定義するフレームのベースからの変位とされます。

セグメントインデックスおよびグループインデックスの双方が0であるときだけ、フレーム番号が存在します。

0以外のグループインデックスは、あるグループを指定します。このグループは、このレコード中で定義されるすべてのパブリックシンボルを、参照のための「参照のフレーム」とします。つまりLINKは、次の動作を行います。

1. 次に示す形式のフィックスアップ

```
TARGET   :   EI(P)
FRAME    :   TARGET
```

これらは（このときの「P」は、このPUBDEFレコード中のパブリックシンボルです）、LINKによって次の形式のフィックスアップに変換されます。

TARGET　：　SI(L)
FRAME　：　GI(G)

このときの「SI(L)」と「d」は、セグメントインデクスと PUBLIC OFFSET フィールドによって与えられます。正常な動作では、新しいフィックスアップ中のフレーム指定子を、古いフィックスアップ（FRAME：TARGET）と同一視します。

2. セグメントインデックス、パブリックオフセットとしてパブリックシンボルの値が定義され、（オプションで）フレーム番号フィールドが｛ベース：オフセット｝の対に変換されるとき、ベース部分は、示されたグループのベースとされます。ここで 0 以外の 16 ビットオフセットが、パブリックシンボル値の定義を満足しないとエラーになります。

グループインデックスが 0 の場合、グループを指定しません。LINK は、シンボルを参照するフィックスアップのフレーム指定を変更することはありません。そして LINK は、これをパブリックシンボルの絶対値のベース部分を、セグメントインデックスフィールドによって決定されるセグメント（LSEG または PSEG）の正規フレームとします。

PUBLIC NAME（パブリック名）

PUBLIC NAME フィールドは、オブジェクトの名前を与えます。そして、そのオブジェクトの MAS 中のロケーションは、他のモジュールで使用可能になります。名前は 1 つ以上の文字を含まなければなりません。

PUBLIC OFFSET（パブリックオフセット）

PUBLIC OFFSET フィールドは 16 ビット値で、LSEG（セグメントインデックス >0 の場合）に対応したパブリックシンボルのオフセット、または指定したフレーム（セグメントインデックス＝ 0 の場合）に対応したパブリックシンボルのオフセットです。

TYPE INDEX（型 INDEX）

TYPE INDEX フィールドは、パブリックシンボルの表す実質の型の記述子を含む単一の前にある TYPDEF（型定義）レコードを識別します。リンカはこのフィールドを無視します。

■エクスターナル名定義レコード（EXTDEF）

REC TYP 8CH (1)	RECORD LENGTH (2)	EXTERNAL NAME (1以上)	TYPE INDEX (1以上)	CHK SUM (1)
		RPT	RPT	

このレコードは、エクスターナル名のリストおよび、各名前について、名前の表現するオブジェクトの型を与えます。LINKは、各エクスターナル名に相当するパブリック名（存在するときは）の与える値を割り当てます。

EXTERNAL NAME（エクスターナル名）

このフィールドは、エクスターナルオブジェクトの名前（長さが0であってはならない）を与えます。

エクスターナル名レコードが名前を含むと、パブリックシンボルとして宣告された同一の名称を含むモジュールに、オブジェクトファイルをリンクするための暗黙の要求になります。この要求は、エクスターナル名が何かのFIXUPPレコードによって参照されるか否かによって発生します。

モジュールでのEXTDEFレコードの順序づけは、各EXTDEFレコード中のエクスターナル名の順序づけとともにモジュールによって要求される、すべてのエクスターナル名配置の順序を発生します。したがって、エクスターナル名は1、2、3、4、…と番号づけされます。この番号は、FIXUPPレコードのTARGET DATUMや、またはFRAME DATUMフィールドの「エクスターナルインデックス」として、特定のエクスターナル名を参照するために使われます。

注意　8086のエクスターナル名は、1、2、3、…と確実に番号がつけられています。この点は8086のエクスターナル（外部）名の番号が0、1、2、…と0から始まっていた点と異なります。これは、特定の意味をもつデフォルト値として0を使う、他の8086インデックス（セグメントインデックス、型インデックス等）を考慮したためです。
エクスターナルインデックスは、前方参照することはありません。たとえばK番目のオブジェクトを定義するエクスターナル定義レコードは、そのオブジェクトをインデックスKで参照するすべてのレコードの前に置かれます。

TYPE INDEX（型インデックス）

このフィールドは、前にあるエクスターナルシンボルによって名前付けされたオブジェクトの型の記述子を含む1つのTYPDEF（型定義）レコードを識別するものです。

LINKでは、型インデックスが共有変数の割り振りにのみ使われます。

■ 行番号レコード（LINNUM）

REC TYP 94H (1)	RECORD LENGTH (2)	LINE NUMBER BASE (1以上)	LINE NUMBER (2)	LINE NUMBER OFFSET (2)	CHK SUM (1)
			RPT		

このレコードは、ソースコード中の行番号と、それに対して翻訳されたコードの対応づけの手段をトランスレータに与えるものです。

LINE NUMBER BASE（行番号ベース）

行番号ベースは、つぎの形式を取ります。

GROUP INDEX (無視される) (1以上)	SEGMENT INDEX (1以上)

セグメントインデックスは、あるソースの行番号に対応する先頭のバイトのロケーションを決定します。

LINE NUMBER（行番号）

0〜32767の行番号を2進法で与えます。高位ビットは、他の目的で使用するために予約されており、0になっています。

LINE NUMBER OFFSET（行番号オフセット）

LINE NUMBER OFFSETフィールドは、16ビット値で行番号LSEGに対応したオフセットです。（セグメントインデックス>0の時）

■ 論理列挙データレコード（LEDATA）

REC TYP A0H (1)	RECORD LENGTH (2)	SEGMENT INDEX (1以上)	ENUMERATED DATA OFFSET (2)	DAT (1)	CHK SUM (1)
				RPT	

このレコードは、8086メモリイメージの一部を構成する連続データを与えます。

SEGMENT INDEX（セグメントインデックス）

このフィールドは0であってはならず（LEDATA RECORDの前に置かれた）、セグメント定義レコードに関するインデックスを指定します。

ENUMERATED DATA OFFSET

このフィールドは、（セグメントインデックスで指定される）LSEGのベースに関してのオフセットを指定し、DATフィールドの第1バイトの相対ロケーションを定義します。DATフィールドの連続したデータバイトは、メモリの高位ロケーションを連続して占めます。

DAT

このフィールドはリロケータブル、または絶対データの連続した（最大1024までの）バイトを与えます。

■ 論理反復データレコード（LIDATA）

REC TYP A2H (1)	RECORD LENGTH (2)	SEGMENT INDEX (1以上)	ITERATED DATA OFFSET (2)	ITERATED DATA BLOCK (1以上)	CHK SUM (1)
				RPT	

このレコードは、8086メモリイメージの一部を構成する連続データを与えます。

SEGMENT INDEX（セグメントインデックス）

このフィールドは0であってはならず、（LIDATAレコードの前に置かれた）SEGDEFレコードに関係するインデックスを指定します。

ITERATED DATA OFFSET

このフィールドは、（セグメントインデックスで指定される）LSEGのベースに関してのオフセットを指定し、ITERATED DATA BLOCKの第1バイトの相対ロケーションを定義します。ITERATED DATA BLOCKの連続したデータバイトは、メモリの高位ロケーションを連続して占めます。

ITERATED DATA BLOCK

このフィールドは、反復するデータバイトを指定するための構造になっています。次に、この構造のフォーマットを示します。

REPEAT COUNT (2)	BLOCK COUNT (2)	CONTENT (1以上)

注意　LINKは、ITERATED DATA BLOCKの大きさが512バイトを超えるLIDATAレコードを扱うことはできません。

REPEAT COUNT

このフィールドは、ITERATED DATA BLOCKのCONTENTの部分の反復回数を指定します。REPEAT COUNTは0であってはなりません。

BLOCK COUNT

このフィールドは、このITERATED DATA BLOCKのCONTENT部にあるITERATED DATAのBLOCK COUNTを指定します。このフィールドの値が0であると、ITERATED DATA BLOCKのCONTENT部はデータバイトとして解釈されます。0以外の場合、CONTENT部にはITERATED DATA BLOCKが、その数だけ繰り返されます。

CONTENT

このフィールドは、前のBLOCK COUNTフィールドの値にしたがって、次の2つの方法のうちの一方で解釈されます。

BLOCK COUNTが0である場合、このフィールドは1バイトのカウントと、そのカウントによって数が示されるデータバイトになります。

BLOCK COUNTが0以外の場合、このフィールドは別のITERATED DATA BLOCKの第1バイトとして解釈されます。

注意　一番外のレベルから数えて、ネスト（入れ子）されているITERATED DATA BLOCKの数は、17以下に制限されています。つまり反復レベル数は、17以下に限定されています。

■ フィックスアップレコード（FIXUPP）

REC TYP 9CH (1)	RECORD LENGTH (2)	THREAD or FIXUP (1以上)	CHK SUM (1)
		RPT	

このレコードは、0かそれ以上のフィックスアップを指定します。各フィックスアップは、前にあるDATAレコード中のロケーションに対して変更（フィックスアップ）を要求します。DATAレコードは、それを参照する1つ以上のフィックスアップレコードを従えることができます。各フィックスアップは、ロケーション、モード、ターゲット、フレームの4つのデータを指定するFIXUPフィールドによって指定されます。フレームとターゲットは、完全にフィックスアップファ

イルが指定されるか、または前の THREAD フィールドを参照することで指定されます。

THREAD フィールドは、ターゲットまたはフレームを識別するために、その後に参照されるデフォルトターゲット、またはデフォルトフレームを指定します。フレーム指定のために 4 つ、ターゲット指定のために 4 つの計 8 つの THREAD（スレッド）が指定されます。スレッドによって、一度ターゲットおよびフレームが指定されると、型（ターゲットまたはフレーム）とスレッド番号（0〜3）は、同一の THREAD フィールドが（同じレコード、または他の FIXUPP レコード中で）現れるまで、後に続く FIXUP フィールドによって参照されます。

THREAD（スレッド）

THREAD フィールドのフォーマットは次のとおりです。

TRD (1)	INDEX (1 以上)
	COND

TRD DAT（ThReaD DATa：スレッドデータ）サブフィールドは、次の内部構造をもつバイトです。

0	D	Z	METHOD	THRED

「Z」は 1 ビットのサブフィールドで、現在、機能が定義されておらず、0 である必要があります。

「D」サブフィールドは、指定されているスレッド型を識別する 1 ビットです。D = 0 の場合、ターゲットスレッドが定義されていますが、D = 1 の場合は、フレームスレッドが定義されています。

METHOD は、0〜3（D = 0 の場合）または 0〜6（D = 1 の場合）を取る 3 ビットのサブフィールドです。

D = 0 の場合、METHOD は（0、1、2、3、4、5、6、7）を 4 で割った余りの値を取ります。ここに、0、……、7 が A.8 に示すターゲットを指定する方法 T0、……、T7 を示します。このように METHOD は、第 1 または第 2 の方法で、ターゲットが指定されたか否かを示すことなく、ターゲットの指定に必要なインデックスやフレーム番号の種類を示します。方法 2b、3、7 は、LINK で使えないことに注意してください。

D = 1 の場合、METHOD = 0、1、2、4、5 は、フレームを指定する方法 F0、……、に対応します。ここで METHOD は、フレームを指定するために必要なインデックス（存在する場合は）の種類を示します。方法 3 と、5d は LINK で使えないことに注意してください。

スレッドは0〜3の数で、スレッドフィールドによって定義されるフレームまたはターゲットのスレッド番号と結びつきます。

インデックスは、METHODサブフィールド中の指定によってセグメントインデックス、グループインデックス、またはエクスターナルインデックスになります。このサブフィールドは、METHODにF4、またはF5が指定されていると存在しません。

FIXUP（フィックスアップ）

次に、FIXUPフィールドのフォーマットを示します。

LOCAT	FIX DAT	FRAME DATUM	TARGET DATUM	TARGET DISPLACEMENT
(2)	(1)	(1以上)	(1以上)	(1以上)
		COND	COND	COND

LOCATは、次のフォーマットをもつ2バイトです。

1	M	S	LOC	DATA RECORD OFFSET
LOBYTE				HIBYTE

「M」はフィックスアップのモード（自己相対（M = 0）、セグメント相対（M = 1））を指定する1ビットサブフィールドです。

注意　LIDATAレコードには、自己相対フィックスアップが適応できない場合もあります。

「S」は、ターゲットDISPLACEMENTサブフィールドの長さを指定する1ビットサブフィールドです。FIXUPフィールド中にTARGET DISPLACEMENTが存在する（以下参照）と、2バイト（16ビットの負でない数、S = 0）または3バイト（24バイト数の2の補数、S = 1）の値を取ります。

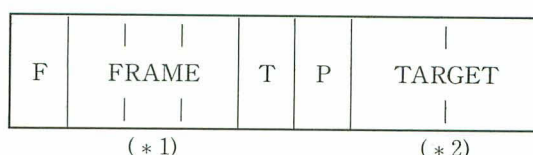
注意　3バイトサブフィールドは、将来の拡張により存在し得ますが、現在は使用されていません。したがって、現在はS = 0に強制されます。

LOCは、フィックスアップされる先行DATAレコード中のバイトが、何であるかを示す3ビットのサブフィールドで、LOC = 0の場合「低位バイト」、LOC = 1の場合「オフセット」、LOC = 2の場合「ベース」、LOC = 3の場合「ポインタ」、LOC = 4の場合「高位バイト」になります。LOCの他の値は無効です。

DATA RECORD OFFSETは0〜1023をとる数で、先行するDATAレコー

ド中の低位バイトのロケーション（フィックスアップされる実際のバイト）の相対位置を与えます。DATA RECORD OFFSET は、DATA レコード中のデータフィールドの第1バイトと対応します。

注意　先行する DATA レコードが LIDATA レコードであると、DATA RECORD OFFSET の値が、ITERATED DATA フィールドの REPEAT COUNT サブフィールド、または BLOCK COUNT サブフィールド中の「ロケーション」を示すこともあります。しかし、このような参照はエラーになります。このような無効レコードに対して、LINK の動作は不定となります。

次に、FIX DAT バイトのフォーマットを示します。

(*1) フレーム指定方法 2b、F3、F5d は使えません。
(*2) ターゲット指定方法 T3、T7 は使えません。

「F」は、このフィックスアップのフレームが、スレッドによって指定される（F = 1）か、または明確に指定するか（F = 0）を与える1ビットサブフィールドです。

フレームは、F ビットで示されるどちらかの方法によって解釈される数です。F が 0 の場合、フレームはフレーム指定方法 F0、……、F5 に対応する 0〜5 の数です。F = 1 の場合、フレームはスレッド番号（0〜3）です。これは、同一スレッド番号のついたフレームスレッドを定義する THREAD フィールドによって、最も最近に定義されたフレームを指定します（THREAD フィールドは、同一の、または先行する FIXUPP レコード中に存在します。）

「T」は、このフィックスアップに指定されるターゲットが、スレッド参照によって定義される（T = 1）か、または FIXUP フィールド中で明確に指定される（T = 0）かを示す1ビットのサブフィールドです。

「P」は、ターゲットが第1の方法で指定される（TARGET DISPLACEMENT が必要、P = 0）か、または第2の方法で指定される（TARGET DISPLACEMENT が不要、P = 1）かを示す1ビットのサブフィールドです。ターゲットスレッドは、第1／第2属性をもたないため、P ビットはターゲット指定の第1／第2属性を与える唯一のフィールドです。

ターゲットは、2ビットのサブフィールドとして解釈されます。T = 0 の場合、ターゲットフィールドは、P の値によって（P は T0、……、T7 の高位ビットとして解釈されます）T0、……、T3、または T4、……、T7 に対応する 0〜3 の数を与えます。スレッドによってターゲットを指定する場合（T = 1）、ターゲット

はスレッド番号（0～3）を指定します。

FRAME DATUM は、フレーム指定の「参照」部で、セグメントインデックス、グループインデックス、エクスターナルインデックスのいずれかです。FRAME DATUM サブフィールドは、フレームがスレッドによって指定されず（F = 0）、方法 F4、F5、F6 によって明示されない場合にのみ存在します。TARGET DATUM は、ターゲット指定の「参照」部で、セグメントインデックス、グループインデックス、エクスターナルインデックスまたはフレーム番号のいずれかです。

TARGET DATUM サブフィールドは、ターゲットがスレッドによって指定されないときだけ（P = 0）存在します。

TARGET DISPLACEMENT は、ターゲットを指定する「第1の」方法が要求する2バイトの変位です。この2バイトサブフィールドは、P = 0 のときだけ存在します。

注意　これらの方法については、すべて A.8「フィックスアップのためのフレームの概念」に解説があります。

■ モジュールエンドレコード（MODEND）

REC TYP 8AH (1)	RECORD LENGTH (2)	MOD TYP (1)	START ADDRS (1以上)	CHK SUM (1)
			COND	

このレコードのオブジェクトは2つあります。このレコードはモジュールの終了を示し、終了したばかりのモジュールに実行開始のエントリポイントが指定されているか否かを示します。後者が存在すると、実行アドレスも指定します。

MOD TYP

このフィールドは、このモジュールの属性を示します。ビット割り当てに関連する意味は次のとおりです。

MATTER	Z	Z	Z	Z	Z	L

MATTER は次に示す、モジュール属性を指定する2ビットサブフィールドです。

MATTER	モジュール特性
0	非メインモジュールにスタートアドレスなし
1	非メインモジュールにスタートアドレスあり
2	メインモジュールにスタートアドレスなし
3	メインモジュールにスタートアドレスあり

「L」は、START ADDRSフィールドがLINKによるフィックスアップが必要な論理アドレスとして解釈される（L＝1）か否かを示します。また、LINKでは、Lが常に1に固定されることに注意してください。

「Z」は、そのビットに現在機能割り付けられていないことを示します。このビットは0である必要があります。

物理開始アドレス（L＝0）は使用できません。

START ADDRSフィールド（MATTERが1、および3のときのみ存在）のフォーマットは次のとおりです。

START ADDRESS

END DAT (1)	FRAME DATUM (1以上)	TARGET DATUM (1以上)	TARGET DISPLACEMENTSUM (2)
	COND	COND	COND

モジュールの開始アドレスは、モジュール中に存在する他の論理参照のすべての属性をもちます。

論理開始アドレスから物理開始アドレスへのマッピングは、他の（フィックスアップやFIXUPPレコードの解説で述べたような）論理開始アドレスから物理アドレスへのマッピングとまったく同様の方法で行われます。START ADDRSフィールドの前述のサブフィールドは、FIXUPPレコード中のFIX DAT、FRAME DATUM、TARGET DATUM、TARGET DISPLACEMENTフィールドと同じ意味をもっています。「第1」フィックスアップのみが許されています。フレーム指定方法F4は認められません。

■コメントレコード（COMENT）

REC TYP 88H (1)	RECORD LENGTH (2)	COMMENT TYPE (2)	COMMENT (1以上)	CHK SUM (1)

このレコードによって、トランスレータは、オブジェクトにコメントを含むことができます。

COMMENT TYPE

このフィールドは、このレコードのもつコメントの型を示します。これによりコメントに対して、選択的に動作するような手段に対して、コメントを構成することができます。このフィールドのフォーマットは次のとおりです。

N P	N L	Z	Z	Z	Z	Z	Z	COMMENT CLASS

NP（NOPURGE：除去なし）ビットが1であると、COMENTレコードを削除するできるオブジェクトファイルユーティリティプログラムでも、除去することができないことを示します。

NL（NOLIST：リストなし）ビットが1であると、オブジェクトCOMENTレコードのリスト機能をもつオブジェクトファイルユーティリティプログラムのリスティングファイル中に、COMMENTフィールドの文章をリストすることができないことを示します。

次に、COMMENT CLASSフィールドの定義を示します。

0	言語トランスレータコメント。
1	インテル著作権コメント。 NPビットを設定しなければなりません。
2〜155	インテルの使用のために予約。 次の「注意1」を参照してください。
156〜255	ユーザーのために予約。 インテル社の製品に対して、これらの中の値は意味をもちません。 次の「注意2」を参照してください。

COMMENT

このフィールドはコメント情報を与えます。

注意1　クラス値129は、リンカのライブラリ検索リストに加えるためのライブラリの指定に使います。この場合、COMMENTフィールドはライブラリ名を含みます。すべての他の名称指定と異なり、ライブラリ名には、その長さが付けられていないことに注意してください。その長さは、レコード長によって決定されます。「NODEFAULTLIBRARYSEARCH」スイッチによって、リンカは、クラス値が129のCOMMENTレコードをすべて無視します。

注意2　クラス値156は、MS–DOSレベル番号の指定に使います。クラス値が156のとき、COMMENTフィールドにはMS–DOSレベル番号を指定する2バイト整数が含まれます。

B.13　レコードの番号によるリスト

*6E	RHEADR
*70	REGINT
*72	REDATA
*74	RIDATA
*76	OVLDEF
*78	ENDREC
*7A	BLKDEF
*7C	BLKEND
*7E	DEBSYM
80	THEADR
*82	LHEADR
*84	PEDATA
*86	PIDATA
88	COMENT
8A	MODEND
8C	EXTDEF
8E	TYPDEF
90	PUBDEF
*92	LOCSYM
94	LINNUM
96	LNAMES
98	SEGDEF
9A	GRPDEF
9C	FIXUPP
*9E	(none)
A0	LEDATA
A2	LIDATA
*A4	LIBHED
*A6	LIBNAM
*A8	LIBLOC
*AA	LIBDIC

注意　(*) がついたレコード型は、LINK で使えません。オブジェクトモジュール中にある場合も、無視されます。

B.14　共有変数の型に関するマイクロソフト表現法

本章は、8086 と 80286（80286 と上位互換性のあるものを含む）上での共有変数割り振りに関するマイクロソフト規格を定義します。

共有変数は、最終サイズと最終ロケーションが、コンパイル時に固定されず、初期化されないパブリック変数です。相互にリンクされる複数のモジュール中に、共有変数が宣言されていて、いくつかの宣言中で指定された最大サイズとその実効サイズが等しいとき、共有変数は FORTRAN の共有ブロックのようなものになります。また、C 言語では、初期化されていないパブリック変数は共有変数です。次に、C 言語による同一の共有変数の異なる宣言の例を示します。

```
char foo[4];    /*  In file a.c   */
char foo[1];    /*  In file b.c   */
char foo[1024]; /*  In file c.c   */
```

a.c、b.c、c.c によって作成されたオブジェクトが相互にリンクされていると、リンカは文字アライメント「foo」に 1024 バイトを割り当てます。

オブジェクトテキストの中で、エクスターナル定義レコード（EXTDEF）と、それが参照する型定義レコード（TYPDEF）によって、共有変数が定義されます。

共有変数に対する TYPDEF のフォーマットは次のとおりです。

REC TYP 8EH (1)	RECORD LENGTH (2)	0 (1)	EIGHT LEAF DESCRIPTOR (1 以上)	CHK SUM (1)

EIGHT LEAF DESCRIPTOR（8 リーフ記述子）フィールドのフォーマットは次のとおりです。

E N (1 以上)	LEAF DESCRIPTOR (1 以上)

EN フィールドは、LEAF DESCRIPTOR フィールド中の次の 8 つのリーフが EASY（単純）であるか（ビット＝ 0）、NICE（精密）であるか（ビット＝ 1）を指定します。共有変数の TYPDEF では、このバイトが常に 0 です。

LEAF DESCRIPTOR フィールドは、次の 2 つのフォーマットのうちいずれかを取ります。

デフォルトのデータセグメント中の（near 変数）共有変数フォーマットは次のとおりです。

NEAR 62H (1)	VAR TYP (1)	LENGTH IN BITS (1以上)	VAR SUBTYP (1以上)
			(OPTIONAL)

VARTYP（変数型）フィールドは、SCALAR（スカラ；7BH）、STRUCT（構造体；79H）またはARRAY（配列；77H）のいずれかです。VAR SUBTYP フィールドは、リンカに無視されます。
デフォルトのデータセグメント中にない共有変数のフォーマットは次のとおりです。

FAR 61H (1)	VAR TYP 77H (1)	NUMBER OF ELEMENTS (1以上)	ELEMENT TYP INDEX (1以上)

この VARTYP（変数型）フィールドは、ARRAY（77H）に限られます。RECORD LENGTH フィールドによって UMBER OF ELEMENTS を指定し、ELEMENT 型インデックスのフォーマットが、その（near）共有変数の形をしている定義済みの TYPDEF のインデックスになります。

LENGTH IN BITS や NUMBER OF ELEMENTS フィールドのフォーマットは、本マニュアルの TYPDEF レコードフォーマットの説明にある LEAF DESCRIPTOR フィールドのフォーマットと同一です。

リンク時間の意味

先行して記述されたフォーマットのうち、1つの TYPDEF を参照する、すべての EXTDEF は、共有変数として扱われます。他は、すべて整合パブリックシンボル定義（PUBDEF）をもつはずのエクスターナル定義シンボルとして扱われます。共有変数定義に整合する PUBDEF は、共有変数をオーバーライドします。2つの共有変数定義は、定義の中で与えられる名前が整合するとき、一致するといいます。共有変数が near、far にかかわらず、2つの整合する定義が一致しないと、リンカは変数が near であると仮定します。

変数が near であると、指定されたサイズのうちで、そのサイズを最大とします。変数が far であると、リンカはアライメント（配列）要素のサイズ指定に矛盾があると、警告を表示します。このような矛盾がなければ、変数のサイズは要素サイズに指定された最大要素数をかけたものになります。すべての near 変数のサイズの合計は、64K バイトを超えることはできません。すべての far 変数のサイズの合計は、その機械のアドレス指定可能メモリ空間を超えることはできません。

「HUGE」共有変数

64K バイトを超えるサイズをもつ far 変数は、連続したセグメント中（8086）か、または連続選択装置（80286）中に置かれます。セグメント中に、huge 共有変数は、他のデータ項目を置きません。
リンカが大きな共有変数と near 共有変数を整合させる定義を見つけると、警告メッセージを発します。near 変数は、64K バイトより大きいことがあり得ないからです。

付録 C

各種コード一覧

この章には、プログラム作成時に役立つ各種コード一覧を収録しています。収録してある表を以下に示します。

・アスキー制御コード表
・アスキー文字コード表
・エスケープシーケンス表
・1バイト/2バイト変換表

■ アスキー制御コード表

16進	文字	名称	略号	内　容
07H	^G	¥a	(BEL)	ベル（BEL）
08H	^H	¥b	FEO(BS)	後退（BS）
09H	^I	¥t	FE1(HT)	水平タブ（HT）
0AH	^J	¥n	FE2(LF)	改行（LF）
0BH	K	¥v	FE3(VT)	垂直タブ（VT）
0CH	^L	¥f	FE4(FF)	書式送り（FF）
0DH	^M	¥r	FE5(CR)	復帰（CR）
1AH	^Z	SUB	S	置換キャラクタ
1BH	^[	ESC	E	拡張
1EH	^^	IS2(RS)	R	レコード分離キャラクタ

■ アスキー文字コード表

上位4ビット→ / 下位4ビット↓

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		DE		0	@	P	‘ (注)	p				ー	タ	ミ		
1	SH	D1	!	1	A	Q	a	q			。	ア	チ	ム		円
2	SX	D2	"	2	B	R	b	r			「	イ	ツ	メ		年
3	EX	D3	#	3	C	S	c	s			」	ウ	テ	モ		月
4	ET	D4	$	4	D	T	d	t			、	エ	ト	ヤ		日
5	EQ	NK	%	5	E	U	e	u			・	オ	ナ	ユ		時
6	AK	SN	&	6	F	V	f	v			ヲ	カ	ニ	ヨ		分
7	BL	EB	'	7	G	W	g	w			ァ	キ	ヌ	ラ		秒
8	BS	CN	(	8	H	X	h	x			ィ	ク	ネ	リ	♠	
9	HT	EM	)	9	I	Y	i	y			ゥ	ケ	ノ	ル	♥	
A	LF	SB	*	:	J	Z	j	z			ェ	コ	ハ	レ	♦	
B	HM	EC	+	;	K	[	k	{			ォ	サ	ヒ	ロ	♣	
C	CL	→	,	<	L	¥	l	¦			ャ	シ	フ	ワ	●	(注)
D	CR	←	-	=	M	]	m	}			ュ	ス	ヘ	ン	○	
E	SO	↑	.	>	N	^	n	~			ョ	セ	ホ	゛		
F	SI	↓	/	?	O	_	o				ッ	ソ	マ	゜		

■エスケープシーケンス表

コード	機　　能
ESC[pl;pcH	カーソルをpl行pcカラムに移動させる
ESC[pl;pcf	同上
ESC=lc	ESC[pl;pcHと同じだが、lとcは16進数で20Hが加えられた値となる。lは行位置、cはカラム位置となる
ESC[pnA	カーソルをpn行上の同一カラム位置に移動させる
ESC[pnB	カーソルをpn行下の同一カラム位置に移動させる
ESC[pnC	カーソルをpn文字右に移動させる
ESC[pnD	カーソルをpn文字左に移動させる
ESC[0J	カーソル位置から最終行の右端までをクリアする
ESC[1J	先頭行の左端からカーソル位置までをクリアする
ESC[2J	画面全体をクリアし、カーソルをホーム位置へ移動させる
ESC*	同上
ESC[0K	カーソル位置から行の右端までをクリアする
ESC[1K	行の左端からカーソル位置までをクリアする
ESC[2K	カーソルが位置する行の左端から右端までをクリアする
ESC[pnM	カーソルが位置する行から下をpn行削除する
ESC[pnL	カーソルが位置する行の上にpn行の空白行を挿入する
ESCD	カラム位置をそのままに、カーソルを1行下に移動させる。カーソルが最終行にある場合は、1行スクロールアップする
ESCE	カーソルを1行下の左端に移動させる。カーソルが最終行にある場合は、1行スクロールアップする
ESCM	カラム位置をそのままに、カーソルを1行上の行に移動させる。カーソルが最終行にある場合は、1行スクロールダウンする
ESC[s	カーソル位置と表示文字の属性をセーブする
ESC[u	ESC[sでセーブした内容をロードする。ESC[sが実行されていない場合には、ホーム位置と属性の規定値が与えられる
ESC[6n	カーソル位置を、コンソール入力直後に知らせる
ESC)0	画面モードを漢字モードにする（規定値）
ESC)3	画面モードをグラフ文字モードにする
ESC[>5l	カーソルを画面に表示させる（規定値）
ESC[>5h	カーソルを画面に表示させない
ESC[>lh	ファンクションキーの内容を画面に表示させない
ESC[>ll	ファンクションキーの内容を画面に表示させる（規定値）
ESC[>3h	画面の表示行数を20行にする（ノーマルモードのみ）
ESC[>3n	画面の表示行数を31行にする（ハイレゾモードのみ）
ESC[>3l	画面の表示行数を25行にする（規定値）
ESC[ps;…;psm	表示文字の属性を設定する
〈psの値〉	〈内容〉
0	規定値
1	ハイライト（モノクロのみ）
2	バーティカルライン
4	アンダーライン

コード	機　能
	〈psの値〉　〈内容〉 5　ブリンク 7　リバース 16（または8）　シークレット（不可視） 30　黒 31（または17）　赤 32（または20）　緑 33（または21）　黄色 34（または18）　青 35（または19）　紫 36（または22）　水色 37（または23）　白 40　黒反転 41　赤反転 42　緑反転 43　黄色反転 44　青反転 45　紫反転 46　水色反転 47　白反転
ESC [Pn;…;Pnp	ESC[に続く最初の1文字に対応するキーに、2番目以降の文字、または文字列を割り当てる
ESC [”string”;p	同上
ESC [Pn;”string”;Pnp	同上

■ PC–H98でのみ使用可能なエスケープシーケンス表

PC–H98では、拡張されたハードウェア機能を利用するために、使用できるエスケープシーケンスが（PC–9801xxやPC–98xxより）増えています。

コード	機　能
ESC [?5h	Enable Extended Attribute Mode 拡張アトリビュートモードにする指示です。このモードでは画面の表示文字の色属性をフォアグラウンドカラー（文字色）とバックグラウンドカラー（背景色）に分けて指定できるようになります。
ESC [?5l	Disable Extended Attribute Mode 標準アトリビュートモードにする指示です。システムの規定値はこのモードであり、拡張アトリビュートモードの使用が終了したら、必ずこのモードに戻してください。

<table>
<tr><th>コード</th><th>機　能</th></tr>
<tr><td>ESC [ps;…;psm</td><td>Character Attribute
表示文字に属性を指示することができます。属性は一度指示すると以降に続く表示文字に適用され、次の属性の指定まで有効です。
パラメータ ps は一度に複数指定できますが、色の指定はその内のひとつにする必要があります。ps には次の値を用いますが、2つのどれかを指定できるものもあります。
<table>
<tr><th>〈ps の値〉</th><th>標準モード</th><th>拡張モード (*1)</th></tr>
<tr><td>0</td><td>規定の属性</td><td>←</td></tr>
<tr><td>1</td><td>ハイライト (*2)</td><td>←</td></tr>
<tr><td>2</td><td>バーティカルライン</td><td>←</td></tr>
<tr><td>4</td><td>アンダーライン</td><td>←</td></tr>
<tr><td>5</td><td>ブリンク</td><td>←</td></tr>
<tr><td>7</td><td>リバース</td><td>←</td></tr>
<tr><td>16（または 8）</td><td>シークレット（不可視）</td><td>←</td></tr>
<tr><td></td><td></td><td>フォアグラウンド</td></tr>
<tr><td>30</td><td>黒　淡（暗）</td><td>←</td></tr>
<tr><td>31（または 17）</td><td>赤</td><td>←</td></tr>
<tr><td>32（または 20）</td><td>緑</td><td>←</td></tr>
<tr><td>33（または 21）</td><td>黄色</td><td>←</td></tr>
<tr><td>34（または 18）</td><td>青</td><td>←</td></tr>
<tr><td>35（または 19）</td><td>紫</td><td>←</td></tr>
<tr><td>36（または 22）</td><td>水色</td><td>←</td></tr>
<tr><td>37（または 23）</td><td>白</td><td>←</td></tr>
<tr><td></td><td></td><td>バックグラウンド</td></tr>
<tr><td>40</td><td>黒反転</td><td>黒</td></tr>
<tr><td>41</td><td>赤反転</td><td>赤</td></tr>
<tr><td>42</td><td>緑反転</td><td>緑</td></tr>
<tr><td>43</td><td>黄色反転</td><td>黄色</td></tr>
<tr><td>44</td><td>青反転</td><td>青</td></tr>
<tr><td>45</td><td>紫反転</td><td>紫</td></tr>
<tr><td>46</td><td>水色反転</td><td>水色</td></tr>
<tr><td>47</td><td>白反転</td><td>白</td></tr>
</table>
規定の属性に戻すには、ESC [m が最適です。
(*1) 拡張モードは、ESC [?5h にて拡張モードにした場合のみ使用可能です。また、拡張モードを使用したプログラムは、終了時にモードを標準モードに戻す必要があります。
(*2) モノクロのみ</td></tr>
</table>

■1バイト/2バイト変換表

		1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	／■ 2223	DE ／■ 2223	2121	0 2330	@ 2177	P 2350	／■ 2223	p 2370	／■ 2223	／■ 2223	／■ 2223	ー 213C	タ 253F	ミ 255F	／■ 2223	／■ 2223
1	SH ／■ 2223	D1 ／■ 2223	! 212A	1 2331	A 2341	Q 2351	a 2361	q 2371	／■ 2223	／■ 2223	。 2123	ア 2522	チ 2541	ム 2560	／■ 2223	円 315F
2	SX ／■ 2223	D2 ／■ 2223	” 2149	2 2332	B 2342	R 2352	b 2362	r 2372	／■ 2223	／■ 2223	「 2156	イ 2524	ツ 2544	メ 2561	／■ 2223	年 472F
3	EX ／■ 2223	D3 ／■ 2223	# 2174	3 2333	C 2343	S 2353	c 2363	s 2373	／■ 2223	／■ 2223	」 2157	ウ 2526	テ 2546	モ 2562	／■ 2223	月 376E
4	ET ／■ 2223	D4 ／■ 2223	$ 2170	4 2334	D 2344	T 2354	d 2364	t 2374	／■ 2223	／■ 2223	、 2122	エ 2528	ト 2548	ヤ 2564	／■ 2223	日 467C
5	EQ ／■ 2223	NK ／■ 2223	% 2173	5 2335	E 2345	U 2355	e 2365	u 2375	／■ 2223	／■ 2223	・ 2126	オ 252A	ナ 254A	ユ 2566	／■ 2223	時 3B7E
6	AK ／■ 2223	SN ／■ 2223	& 2175	6 2336	F 2346	V 2356	f 2366	v 2376	／■ 2223	／■ 2223	ヲ 2572	カ 252B	ニ 254B	ヨ 2568	／■ 2223	分 4A2C
7	BL ／■ 2223	EB ／■ 2223	’ 2147	7 2337	G 2347	W 2357	g 2367	w 2377	／■ 2223	／■ 2223	ァ 2521	キ 252D	ヌ 254C	ラ 2569	／■ 2223	秒 4943
8	BS ／■ 2223	CN ／■ 2223	(214A	8 2338	H 2348	X 2358	h 2368	x 2378	／■ 2223	／■ 2223	ィ 2523	ク 252F	ネ 254D	リ 256A	♠／■ 2223	／■ 2223
9	HT ／■ 2223	EM ／■ 2223	) 214B	9 2339	I 2349	Y 2359	i 2369	y 2379	／■ 2223	／■ 2223	ゥ 2525	ケ 2531	ノ 254E	ル 256B	♥／■ 2223	／■ 2223
A	LF ／ 変換しない	SB ／■ 2223	* 2176	: 2127	J 234A	Z 235A	j 236A	z 237A	／■ 2223	／■ 2223	ェ 2527	コ 2533	ハ 254F	レ 256C	♦／■ 2223	／■ 2223
B	HM ／■ 2223	EC ／ 変換しない	+ 215C	; 2128	K 234B	[214E	k 236B	{ 2150	／■ 2223	／■ 2223	ォ 2529	サ 2535	ヒ 2552	ロ 256D	♣／■ 2223	／■ 2223
C	CL ／■ 2223	→ 222A	, 2124	< 2163	L 234C	¥ 216F	| 236C	¦ 2143	／■ 2223	／■ 2223	ャ 2563	シ 2537	フ 2555	ワ 256F	● 217C	／■ 2223
D	CR ／ 変換しない	← 222B	− 215D	= 2161	M 234D	] 214F	m 236D	} 2151	／■ 2223	／■ 2223	ュ 2565	ス 2539	ヘ 2558	ン 2573	○ 217B	／■ 2223
E	SO ／■ 2223	↑ 222C	2125	> 2164	N 234E	∧ 2130	n 236E	~ 2141	／■ 2223	／■ 2223	ョ 2567	セ 253B	ホ 255B	゛ 212B	／■ 2223	／■ 2223
F	SI ／■ 2223	↓ 222D	／ 213F	? 2129	O 234F	_ 2132	o 236F	／■ 2223	／■ 2223	／■ 2223	ッ 2543	ソ 253D	マ 255E	゜ 212C	／■ 2223	／■ 2223

索 引

英数字

ア

カ

サ

タ

ナ

ハ

マ

ヤ

ラ

ワ